CUI

Lab Manual

CSC270-Database Systems-I

Department of Computer Science
Islamabad Campus

Lab Contents:

Introduction to Oracle and MongDB Servers and their Installations; Querying Database using Relational Algebra; Retrieving Data Using the SQL SELECT Statement; Restricting and Sorting Data; Using Single Row Functions to Customize Output; Reporting Aggregated Data Using the Group Functions; Displaying Data from Multiple Tables Using Joins; Using Subqueries to Solve Queries; Using the Set Operators; Manipulating Data using DML Statements; Using DDL Statements to Create and Manage Tables Indexes, Views, and Data Control Language; Introduction to MongoDB and JSON Format; Retrieving Data using MongoDB API; Manipulating Data using MongoDB API; MongoDB's Aggregation Framework.

Student Outcomes (SO)

S.#	Description
2	Identify, formulate, research literature, and solve complex computing problems reaching substantiated conclusions using fundamental principles of mathematics, computing sciences, and relevant domain disciplines
4	Create, select, adapt and apply appropriate techniques, resources, and modern computing tools to complex computing activities, with an understanding of the limitations
5	Function effectively as an individual and as a member or leader in diverse teams and in multi-disciplinary settings.
9	Recognize the need, and have the ability, to engage in independent learning for continual development as a computing professional

Intended Learning Outcomes

Sr.#	Description	Blooms Taxonomy Learning Level	SO
CLO-5	Apply data processing operations on both relational and non-relational DBMS	*Applying*	4,9
CLO-6	Develop a database system for medium size enterprise in a team environment	*Creating*	2,4,5,9

Lab Assessment Policy

The lab work done by the student is evaluated using rubrics defined by the course instructor, viva-voce, project work/performance. Marks distribution is as follows:

Assignments	Lab Mid Term Exam	Lab Project	Total
25	25	50	100

Note: Midterm exam must be computer based.

List of Labs

Lab 0

Introduction to Oracle and MongDB Servers and their Installations

In this course, we are going to learn and practice concepts related to both relational and non-relational database management systems (DBMS). There are various commercial DBMS available for both of them that are being widely used by developers to manage the data. In this lab manual, to demonstrate various concepts, we are going to use Oracle DBMS for relational part and MongoDB for the non-relational part. The Labs 1-11 covers the relational aspect of the course while Labs 12-14 covers the non-relational aspect of the course.

Tools/Software Requirements for Relational Part

Following are the software that we will need for the relational part of this lab manual:

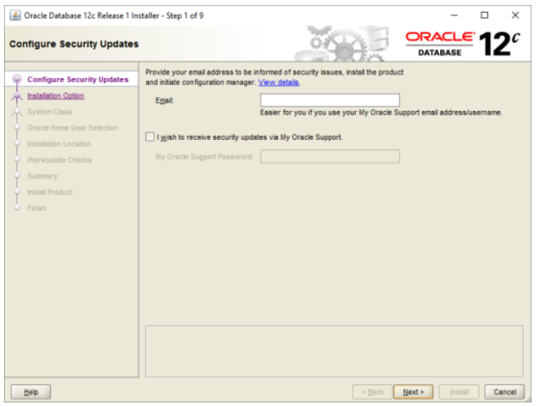
- Oracle Server
- Oracle SQL Developer

Oracle Server

Introduction

The Oracle Server is a relational database management system that provides an open, comprehensive, and integrated approach to information management. It can be used on a local machine such as desktop computer/laptop and on cloud computer.

Oracle Installation



Provide your email address to be informed of security issues, install the product and initiate configuration manager. View details.

Email:

Easier for you if you use your My Oracle Support email address/username.

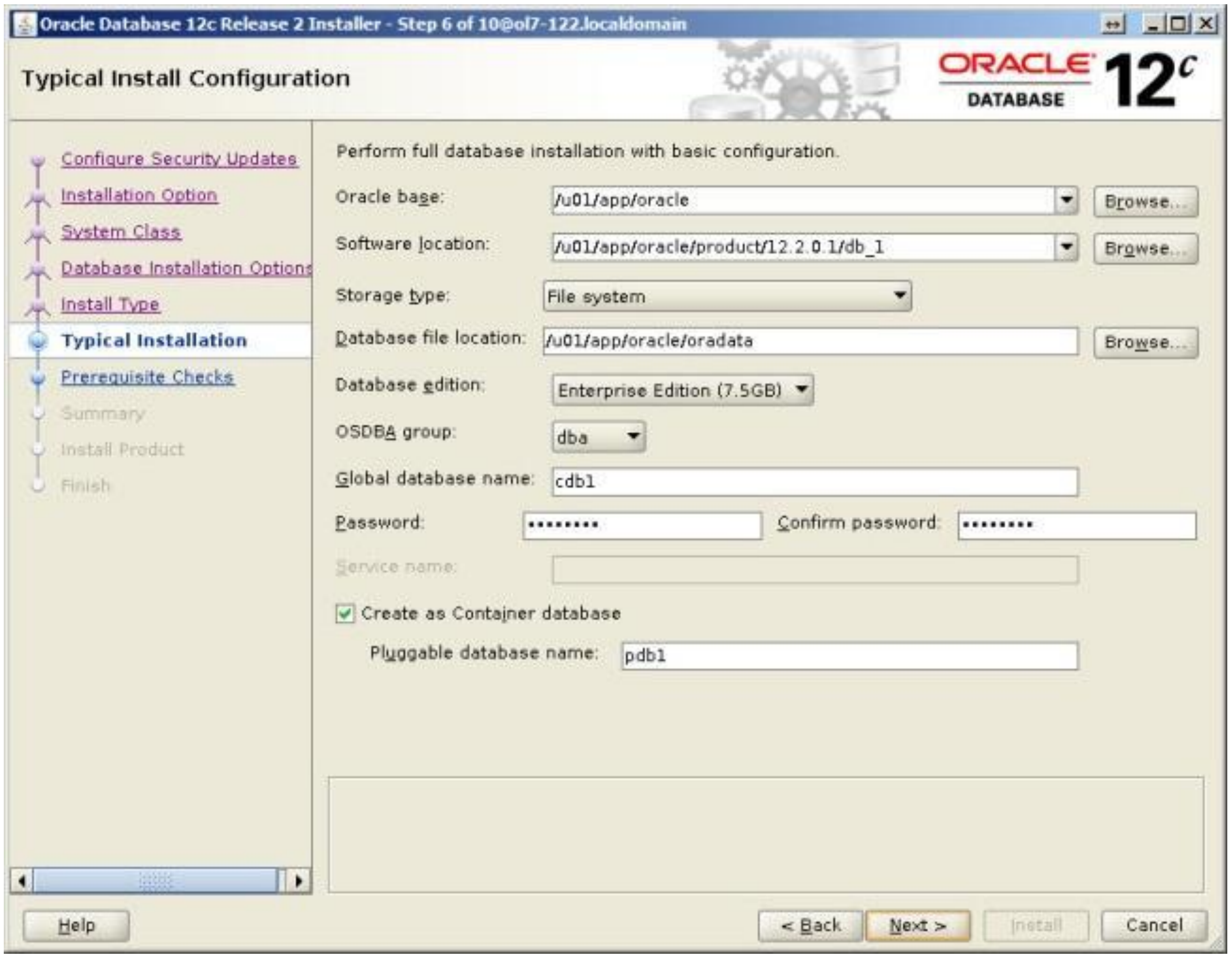
[ ] I wish to receive security updates via My Oracle Support.

My Oracle Support Password:

A free version of Oracle Server can be download by registering at oracle corporation's website. The current version of oracle is 21, while other common versions are 12, 18, and 19. There is no other version between 12 and 18. Both are similar except the version numbering is changed to year of release. The version 12 was released in 2014 where as the version 18, was released in 2018. For the purpose of this course lab work we will be using Oracle 12. You are encouraged to use the latest version and the installation procedure will remain the same.

To install Oracle database for Windows operating system:

1. Go to Oracle Database Software Downloads in your browser.
2. Download the 64-bit .zip file. Select the.zip file and right click to select Extract All. Extract both the .zip files to the same folder.


Oracle Database 12c Release 2 Installer - Step 6 of 10@ol7-122.localdomain

Typical Install Configuration

Perform full database installation with basic configuration.

Oracle base: /u01/app/oracle

Software location: /u01/app/oracle/product/12.2.0.1/db_1

Storage type: File system

Database file location: /u01/app/oracle/oradata

Database edition: Enterprise Edition (7.5GB)

OSDBA group: dba

Global database name: cdb1

Password: ••••••••• Confirm password: •••••••••

Service name:

[✓] Create as Container database

Pluggable database name: pdb1

3. Run the setup.exe and select the installation options according to your database and Windows user requirements.
4. In the Specify Database Identifiers screen of the installation process, enter the Global database name (for example, amc2) and the Oracle system identifier, SID (for example, amc2). Don't select the check box for the option Create as Container database. The Install button gets enabled.
5. Click Install to install the product. Oracle Database is installed on Windows.
6. Start the SQL Plus application. From the command-line, enter the command SQLPLUS to start SQL Plus.

To configure Oracle database on Windows:

1. Log in to SQL*Plus application with sys as sysdba and the password you opted during the installation process in the Schema Password step.
2. Create the user (for example, amc2) and grant access to the database (for example, amc2). The database name is the one that you set while installation.

```
SQL> CREATE USER amc2 IDENTIFIED BY amc2
DEFAULT TABLESPACE users
QUOTA UNLIMITED ON users PASSWORD EXPIRE;

SQL> CREATE ROLE amc2_role;

SQL> GRANT CREATE SESSION, CREATE TABLE, CREATE SEQUENCE, CREATE VIEW,
CREATE TRIGGER to amc2_role;

SQL> GRANT amc2_role TO amc2;
```

Configure your Oracle database QUOTA to UNLIMITED to ensure that enough database storage is available to support large BLOB entries, such as MSI binaries. If you encounter an issue with the SQL Create User statement, then log out of SQL*Plus application and repeat step 1 and step 2.

3. After you successfully create the user, exit the SQL*Plus application and log back into SQL*Plus as user (for example, amc2). You are prompted to set up the password. Set up a strong password.

HumanResources (HR) Schema Description

The Human Resources (HR) schema is a part of the Oracle Sample Schemas that can be installed in an Oracle Database. In this lab manual, we are going to use data from the HR schema.

Table Descriptions

- REGIONS contains rows that represent a region such as America, Asia, and so on.
- COUNTRIES contains rows for countries, each of which is associated with a region.
- LOCATIONS contains the specific address of a specific office, warehouse, or production site of a company in a particular country.
- DEPARTMENTS shows details about the departments in which the employees work. Each department may have a relationship representing the department manager in the EMPLOYEES table.
- EMPLOYEES contains details about each employee working for a department. Some employees may not be assignedto any department.
- JOBS contains the job types that can be held by each employee.
- JOB_HISTORY contains the job history of the employees. If an employee changes departments within a job or changes jobs within a department, a new row is inserted into this table with the earlier job information of the employee.

Guidelines for Installing HR Schema

All scripts necessary to create the Human Resource (HR) schema reside in $ORACLE_HOME/demo/schema/human_resources.
You need to call only one script, hr_main.sql, to create all the objects and load the data. The following steps provide a summary of the installation process:

1. Log on to SQL*Plus as SYS and connect using the AS SYSDBA privilege.

```
sqlplus connect sys as sysdba
Enter password: password
```

2. To run the hr_main.sql script, use the following command:

```
SQL> @?/demo/schema/human_resources/hr_main.sql
```

3. Enter a secure password for HR

```
Specify password for HR as parameter 1:
Enter value for 1:
```

4. Enter an appropriate tablespace, for example, users as the default tablespace for HR

```
Specify default tablespace for HR as parameter 2:
Enter value for 2:
```

5. Enter temp as the temporary tablespace for HR

```
Specify temporary tablespace for HR as parameter 3:
Enter value for 3:
```

6. Enter your SYS password

```
Specify password for SYS as parameter 4:
Enter value for 4:
```

7. Enter the directory path, for example, $ORACLE_HOME/demo/schema/log/, for your log directory

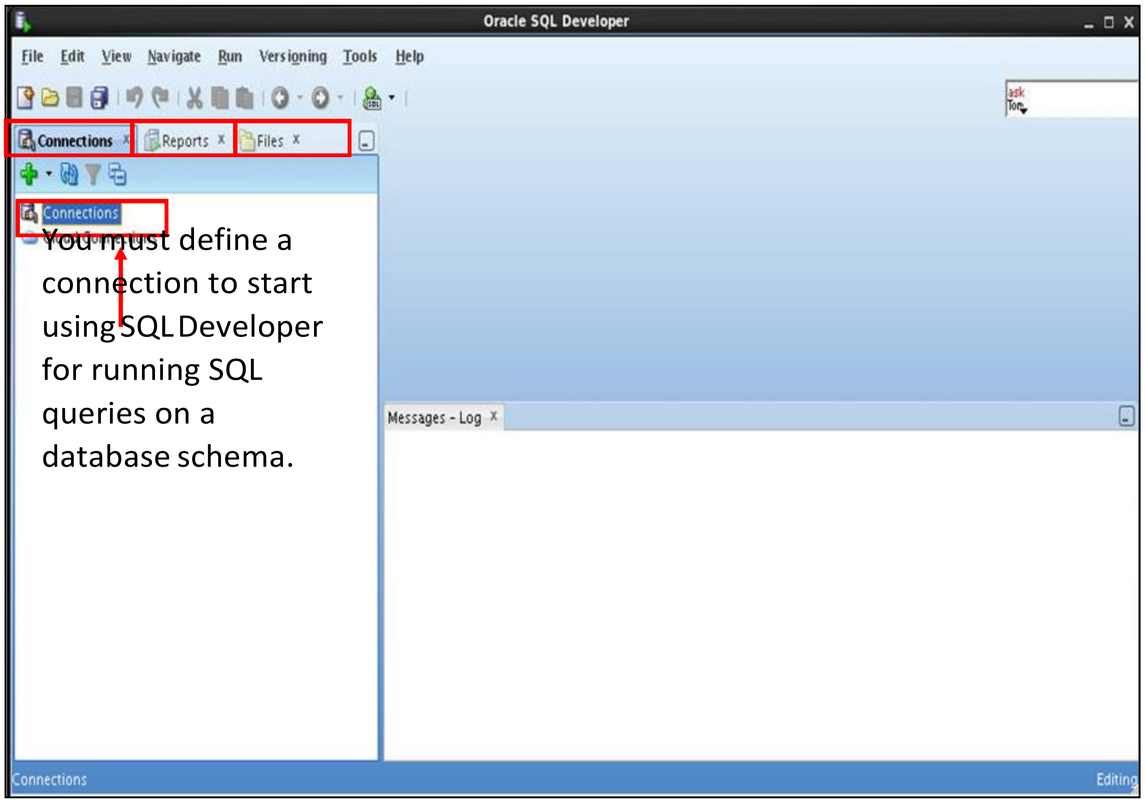
```
Specify log path as parameter 5:
Enter value for 5:
```

Oracle SQL Developer

You can use the SQL*Plus software, that comes with Oracle Server, to interact with Oracle Server for data management tasks or you can install Oracle SQL Developer software.
Oracle SQL Developer is a new, free graphical tool that enhances productivity and simplifies database development tasks. With SQL Developer, you can browse database objects, run SQL statements and SQL scripts, and edit and debug SQL statements. You can also run any number of provided reports, as well as create and save your own. It can be downloaded freely from Oracle's website at https://www.oracle.com/database/technologies/appdev/sqldeveloper-landing.html
SQL Developer, which is the visual tool for database development, simplifies the following tasks:

- Browsing and managing database objects
- Executing SQL statements and scripts
- Editing and debugging PL/SQL statements
- Creating reports

Here's the screenshot of the SQL Developer Interface:


Oracle SQL Developer
File Edit View Navigate Run Versioning Tools Help
Connections
Reports
Files
Connections
You must define a connection to start using SQL Developer for running SQL queries on a database schema.
Messages - Log
Connections
Editing

You need to define at least one connection to be able to connect to a database schema and issue SQL queries or run procedures and functions. To create a database connection, perform the following steps:

1. On the Connections tabbed page, right-click Connections and select New Connection.
2. In the New/Select Database Connection window, enter the connection name. Enter the username and password of the schema that you want to connect to.
    a. From the Role drop-down list, you can select either default or SYSDBA. (You choose SYSDBA for the sys user or any user with database administrator privileges.)
    b. You can select the connection type as Basic. In this type, enter host name and SID for the database that you want to connect to. Port is already set to 1521. You can also choose to enter the Service name directly if you use a remote database connection.

Tools/Software Requirements for Non-Relational Part

Following are the software that we will need for the non-relational part of this lab manual:

- MongoDB Enterprise Server
- Mongo Shell
- MongoDB Compass

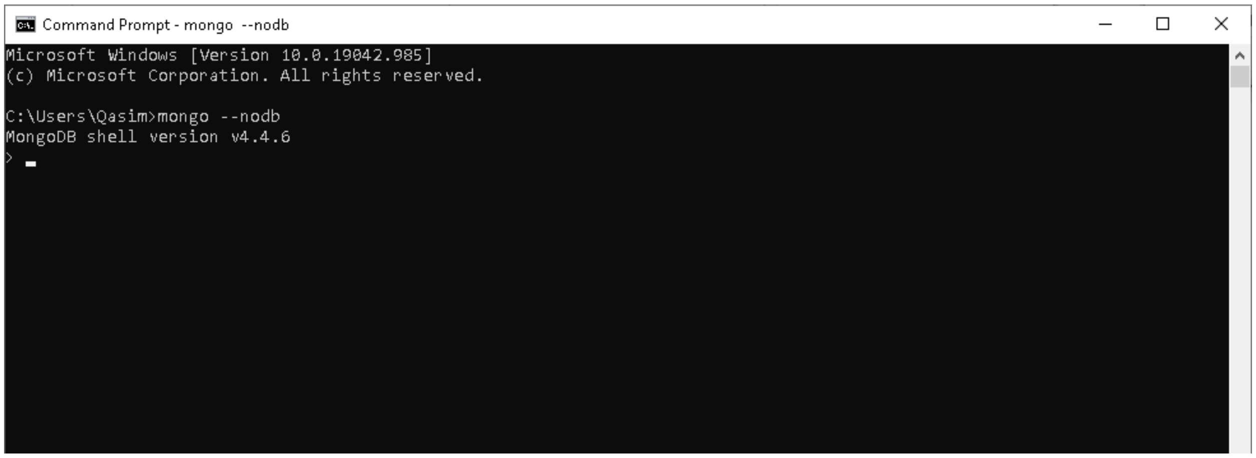
Installation

MongoDB is a non-relational or a NoSQL document-oriented database management system. It is developed by MongoDB Inc. and uses JSON-like documents with optional schemas. MongoDB

Enterprise Server can be downloaded and installed from MongoDB's Download Center at https://www.mongodb.com/try/download/enterprise.
The downloaded installer guides you through the installation process. If you choose the Custom installation option, you may specify an installation directory. MongoDB does not have any other system dependencies. You can install and run MongoDB from any folder you choose. After the installation, make sure to add the path of the bin directory (e.g. C:\Program Files\MongoDB\Server\4.2\bin) to the environment variables. To check if the server is installed correctly, run *cmd* in Windows environment and type *mongo*. In case of successful installation, you should be able to see the messages showing successful connection of Mongo Shell with the locally installed MongoDB server.
The Mongo Shell and MongoDB Compass come pre-included with the server and get installed along with the server. Mongo shell is an interactive command line interface to MongoDB. You can use the Mongo Shell to query and update data as well as perform administrative operations. MongoDB Compass, on the other hand, provide a graphical user interface for querying, aggregating, and analyzing your MongoDB data in a visual environment.
To check if everything installed correctly, please open the command prompt in Windows or Terminal in Linux, and run the following:

```
mongo –nodb
```

If as a result, your MongoDB database version is displayed, this means everything installed correctly.



```
Microsoft Windows [Version 10.0.19042.985]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Qasim>mongo --nodb
MongoDB shell version v4.4.6
>
```

Lab 01

Querying Database using Relational Algebra

Objective:

The objective of this lab is to introduce the Relational Algebra language which forms underlying basis of SQL. It is based on algebra whose operands are relations. Operators are designed to do the most common things that we need to do with relations in a database. This results in an algebra that can be used as a query language for relations.

Activity Outcomes:

After completing this lab, students should be able to do the following:

- Implement relational algebra operations
- Design query expressions by composing relational algebra operations
- Retrieve data from a database using relational algebra expressions

Instructor Note:

For this lab, we are going to use RelaX, a web based relational algebra calculator available at https://dbis-uibk.github.io/relax/landing. It provides a simple single form GUI for querying relational database by writing relational algebra expressions. On top panel it shows all the operators and on left side it shows the name of relations in currently selected database. We can click and select any operation and write/select the attributes of relations.

1) Useful Concepts

Relational algebra takes instances of relations as input and yields instances of relations as output. It uses operators to perform queries. An operator can be either unary or binary. They accept relations as their input and yield relations as their output. Relational algebra is performed recursively on a relation and intermediate results are also considered relations.
The fundamental/core operations of relational algebra are as follows:

- Select (σ)
  - Syntax: σ *condition* (A)
- Project (π)
  - Syntax: π attributes (A)
- Cartesian product (×)
  - Syntax: A × B
- Set difference (-)
  - Syntax: A - B
- Set Union (∪)
  - Syntax: A ∪ B
- Rename (ρ)
  - Syntax: ρ X (A)
  - Syntax: ρ y←a (A)

Following are the derived relational algebra operator:

- Set Intersection (∩)
  - Syntax: A ∩ B
- Natural Join (⋈)
  - Syntax: A ⋈ B
- Theta Join ($⋈_\ominus$)
  - Syntax: A $⋈_\ominus$ B

While forming the condition to be used in *Select* (σ) or *Theta Join* ($⋈_\ominus$) operators, the following comparison and logical operators can be used:

Comparison Operators	
Symbol	**Meaning**
=	Equal
≠	Not Equal
>	Greater Than
≥	Greater Than or Equal To
<	Less Than
≤	Less Than or Equal To

Logical Operators	
Symbol	**Meaning**
∧	And
∨	Or

¬	Not

There are other operators too in extended relational algebra but they are out of the scope for this lab. In this lab, we are going to demonstrate the use of above-mentioned operators over the *IMDB-sample* database already provided in RelaX application.

2) Solved Lab Activites

Activity No	*Allocated Time*	*Level of complexity*	*CLO Mapping*
Activity 1	***2 mins***	***Low***	***CLO-5***
Activity 2	***2 mins***	***Low***	***CLO-5***
Activity 3	***4 mins***	***Low***	***CLO-5***
Activity 4	***6 mins***	***Medium***	***CLO-5***
Activity 5	***15 mins***	***High***	***CLO-5***
Activity 6	***15 mins***	***High***	***CLO-5***
Activity 7	***4 mins***	***Low***	***CLO-5***
Activity 8	***8 mins***	***Medium***	***CLO-5***
Activity 9	***8 mins***	***Medium***	***CLO-5***

Activity 1:

Find the list of all movies which are released after the year 2000

```
σ year > 2000 (movies)
```

Output

This lab activity demonstrates the use of SELECT operator. It takes one operand that is some relation and a condition based on which it filters out the rows of the relation.

Activity 2:

Find all the movies released after 2000 and have a rank greater than 6

```
σ year > 2000 ∧ rank > 6 (movies)
```

Output

This lab activity demonstrates the use of SELECT operator to further filter out the rows based on further complex criteria.

Activity 3:

Slice out the ID column

```
π name,year,rank (σ year > 2000 ∧ rank > 6 (movies))
```

Output

This lab activity deomonstrates the use of PROJECT operator to get rid of ID column from the last activity as it is not providing us with any useful information.

Activity 4:

Find name, year, rank, and genres the movies which are released after the year 2000 and have a rank greater than 6

```
π name,year,rank,genre (σ year > 2000 ∧ rank > 6 (σ id = movie_id (movies ×
movies_genres)))
```

Output

This lab activity demonstrates the use of CARTESIAN PRODUCT. Since the given query involves attributes from two tables, so we need to join the tables before applying the *Select* operator.

Activity 5:

Find name, year, rank, director's first name and last name, and genres of movies which are released after the year 2000 and have a rank greater than 6

```
π name,year,rank,genre,first_name,last_name ( σ year > 2000 ∧ rank > 6 ( σ
directors.id = movies_directors.director_id (( σ id =
movies_directors.movie_id ( (σ id = movie_id (movies × movies_genres)) ×
movies_directors)) × directors)))
```

Output

This lab activity also demonstrates the use of CARTESIAN PRODUCT. This time, however, the given query involves attributes from four tables, so we need to join them appropriately.

Activity 6:

List of actors who never acted in an action movie

```
π first_name, last_name (actors)
-
```

```
π actors.first_name, actors.last_name σ genre='Action' ((σ roles.actor_id =
actors.id (roles × actors)) ⋈ movies_genres)
```

Output

This lab activity demonstrates the use of DIFFERENCE operator together with NATURAL JOIN operator.

Activity 7:

List of actors who have also directed a movie

```
π first_name, last_name actors
∩
π first_name, last_name  directors
```

Output

This lab activity demonstrates the use of INTERSECTION operator.

Activity 8:

List the first and last names of all the directors who are also actors. Use Natural Join.

```
(π first_name, last_name (directors)) ⋈ (π first_name, last_name (actors))
```

Output

This lab activity demonstrates the use of NATRUAL JOIN operator. It's the same query as in the last activity but here we are going to use NATURAL JOIN operator to answer it.

Activity 9:

List the first and last names of all the actors who have played the role of Doctor. Use Natural Join.

```
π actors.first_name,actors.last_name (σ roles.role='Doctor' ((ρid←actor_id
(roles)) ⋈ actors))
```

Output

This lab activity demonstrates the use of RENAME operator. This query can be answered by joining relations using CARTESIAN PRODUCT operator but we instead opted to answer it using NATURAL JOIN operator so as to demonstrate the use of RENAME operator.

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Lab Task 1

Write Relational algebra expressions for the following information needs over IMDB-sample Database:

- *List the first and last names of all the female actors.*
- *List movie names along with their director names of all the movies with a rank greater than 8.5*
- *List titles of all the movies that are released after 2000, have a rank greater than 8, and that belong to Action genre.*
- *List the first and last names of all the actors who played a role in the movie Reservoir Dogs, and the roles they played in it.*
- *List the first and last names of all the actors who acted in the movies of director Quentin Tarantino but not in the movies of director Stanley Kubrick.*

Lab Task 2

Write Relational algebra expressions for the following information needs over University Database:

- *Retrieve the title of the course that is pre-req of 'Database System Concepts'*
- *Retrieve the semester and the year in which 'Einstein' taught the course 'Physical Principles'.*
- *Retrieve the ID and the title of all courses taken by 'Shankar' in 'Fall 2009'.*
- *List the name of students who did not take any course in 'Fall 2009'.*
- *Find building, room number, and capacity of all classrooms in which student 'Tanaka' took all his classes.*

Lab 02

Retrieving Data Using the SQL SELECT Statement

Objective:

In order to retrieve data from the database, we need to use the SQL SELECT statement. Sometimes we may need to retrieve all the columns of a table while the other times we may need to restrict the columns. This lab describes the SELECT statement that is needed to perform these actions.

Activity Outcomes:

After completing this lab, students should be able to do the following:

- List the capabilities of SQL SELECT statements
- Execute a basic SELECT statement
- Able to select all as well as specific columns of a table
- Write SQL statements column heading defaults
- Use of arithmetic expressions and NULL values in the SELECT statement
- Define and use column aliases
- Use of concatenation operator, literal character strings, alternative quote operator, and the DISTINCT keyword
- Display the table structure using the DESCRIBE command

Instructor Note:

Please see Preamble of this lab manual to get a quick introduction about SQL Developer. SQL Developer is a client application that will allow us to connect to Oracle Server, send SQL statements to Oracle Server, get response from Oracle Server, and display the well-formatted results of the queries.
We will be using HR database in majority of the labs to demonstrate the working of various SQL statements. HR database has 7 tables. Please also go through the HR schema detailed in Preamble and get yourself familiarize with it.
Throughout this lab manual, the words keyword, clause, and statement are used as follows:

- A keyword refers to an individual SQL element—for example, SELECT and FROM are keywords.
- A clause is a part of a SQL statement—for example, SELECT employee_id, last_name, and so on.
- A statement is a combination of two or more clauses—for example, SELECT * FROM employees.

1) Useful Concepts

A SELECT statement retrieves information from the database. With a SELECT statement, you can do the following:

- *Projection*: Selects the columns in a table that are returned by a query. Selects a few or as many of the columns as required.
- *Selection*: Selects the rows in a table that are returned by a query. Various criteria can be used to restrict the rows that are retrieved.
- *Joins*: Brings together data that is stored in different tables by specifying the link between them. This will be covered in detail in Lab-6.

Here is the basic form of SELECT statement:

```
SELECT     *|{[DISTINCT] column|expression [alias],...}
FROM       table
[WHERE     logical expression(s)];
```

In its simplest form, a SELECTstatement must include the following:

- A SELECT clause, which specifies the columns to be displayed. It does the *Projection*.
- A FROM clause, which identifies the table containing the columns that are listed in the SELECT and WHERE clause.
- An optional WHERE clause, which is used to restrict the rows to be retrieved from the table specified in the FROM clause. It does the *Selection*. It contains a condition that must be met and it directly follows the FROM clause. If the condition is true, the row meeting the condition is returned.
- Here is what is meant by remaining items in the basic form of SELECT statement:

Item	**Meaning**
*	Selects all columns
DISTINCT	Suppresses duplicates
column|expression	Selects the named column or the expression
alias	Gives different headings to the selected columns
logical expression	Is composed of column names, constants, and a comparison operator. It specifies a combination of one or more expressions and Boolean operators, and returns a value of TRUE, FALSE, or UNKNOWN

2) Solved Lab Activites

Activity No	*Allocated Time*	*Level of complexity*	*CLO Mapping*
Activity 1	***5 mins***	***Low***	***CLO-5***
Activity 2	***5 mins***	***Low***	***CLO-5***
Activity 3	***5 mins***	***Low***	***CLO-5***

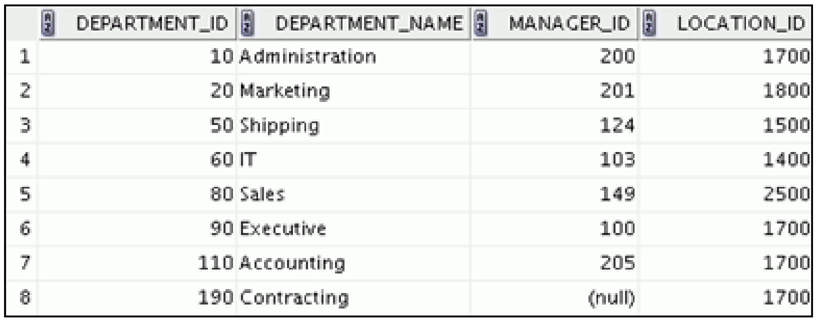
Activity 4	***5 mins***	***Low***	***CLO-5***
Activity 5	***5 mins***	***Low***	***CLO-5***
Activity 6	***5 mins***	***Low***	***CLO-5***
Activity 7	***5 mins***	***Low***	***CLO-5***
Activity 8	***5 mins***	***Low***	***CLO-5***
Activity 9	***5 mins***	***Low***	***CLO-5***
Activity 10	***5 mins***	***Low***	***CLO-5***

Activity 1: Selecting All Columns

Write a SQL statement to select all the columns and all the rows of departments table.

```
SELECT      *
FROM        departments;
```

Output



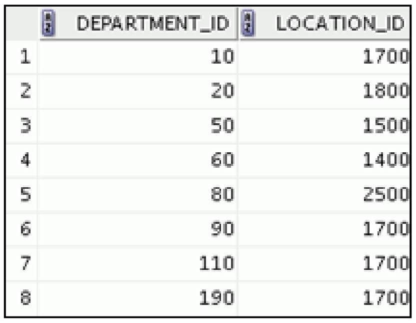
	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	10	Administration	200	1700
2	20	Marketing	201	1800
3	50	Shipping	124	1500
4	60	IT	103	1400
5	80	Sales	149	2500
6	90	Executive	100	1700
7	110	Accounting	205	1700
8	190	Contracting	(null)	1700

Activity 2: Selecting Specific Columns

Write a SQL statement to select only department_id and location_id columns and all the rows of departments table.

```
SELECT    department_id,location_id
FROM      departments;
```

Output



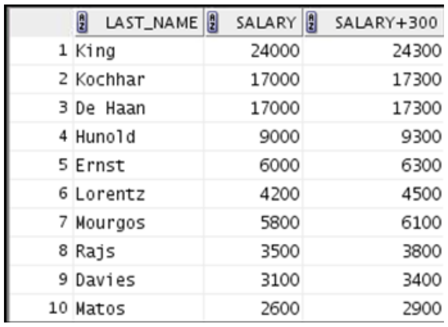
	DEPARTMENT_ID	LOCATION_ID
1	10	1700
2	20	1800
3	50	1500
4	60	1400
5	80	2500
6	90	1700
7	110	1700
8	190	1700

Activity 3: Using Arithmetic Operators

Write a SQL statement to select last_name, salary columns of employees table and another column derived from salary calculating a salary increase of 300.

```
SELECT   last_name, salary, salary + 300
FROM     employees;
```

Output



	LAST_NAME	SALARY	SALARY+300
1	King	24000	24300
2	Kochhar	17000	17300
3	De Haan	17000	17300
4	Hunold	9000	9300
5	Ernst	6000	6300
6	Lorentz	4200	4500
7	Mourgos	5800	6100
8	Rajs	3500	3800
9	Davies	3100	3400
10	Matos	2600	2900

…

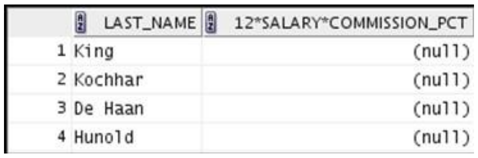
The above statement uses the addition operator to calculate a salary increase of 300 for all employees. It also displays a SALARY+300 column in the output. Note that the resultant calculated column, SALARY+300, is not a new column in the EMPLOYEES table; it is for display only. By default, the name of a new column comes from the calculation that generated it i.e. in this case, salary+300.
Multiply (*), Divide (/), and subtraction (-) are the other operators that can be used in the arithmetic expression.

Activity 4: NULL values in Arithmetic Expression

Write a SQL statement to select last_name of employees table and another column derived from salary and commission_pct calculating the annual commissions of employees.

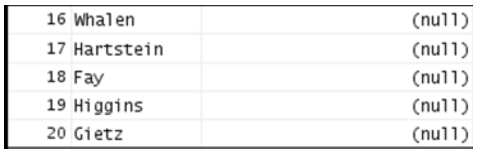
```
SELECT   last_name, 12*salary*commission_pct
FROM     employees;
```

Output



	LAST_NAME	12*SALARY*COMMISSION_PCT
1	King	(null)
2	Kochhar	(null)
3	De Haan	(null)
4	Hunold	(null)

…



```
16 Whalen        (null)
17 Hartstein     (null)
18 Fay           (null)
19 Higgins       (null)
20 Gietz         (null)
```

…

If we look at the COMMISSION_PCT column in the EMPLOYEES table, we will notice that only a sales manager or sales representative can earn a commission. Other employees are not entitled to earn commissions. For the rest of the employees, it will be NULL.
In SQL, NULL is a value that is unavailable, unassigned, unknown, or inapplicable. Null is not the same as zero or a blank space. Zero is a number and blank space is a character.

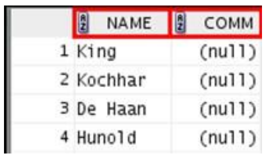
In SQL, arithmetic expressions containing a NULL value evaluate to NULL. Consequently, we can see that the expression evaluates to NULL for all those employees whose COMMISSION_PCT is NULL.

Activity 5: Using Column Aliases

Write a SQL statement to select last_name and commission_pct of employees table. Rename last_name column to name and commission_pct column to comm

```
SELECT    last_name AS name, commission_pct comm
FROM      employees;
```

Output



```
     NAME      COMM
1 King       (null)
2 Kochhar    (null)
3 De Haan    (null)
4 Hunold     (null)
```

…

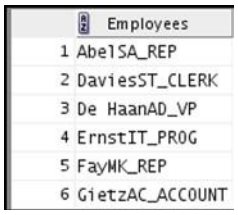
When displaying the result of a query, SQL Developer normally uses the name of the selected column as the column heading. This heading may not be descriptive and, therefore, may be difficult to understand. You can change a column heading by using a column alias. For that you can either specify the alias after the column in the SELECT list using blank space as a separator or using AS keyword between the column name and the alias.

Activity 6: Using Concatenation Operator

Write a SQL statement to select last_name and job_id of employees, concatenate them, and display in a single column.

```
SELECT     last_name||job_id AS"Employees"
FROM       employees;
```

Output



```
  Employees
1 AbelSA_REP
2 DaviesST_CLERK
3 De HaanAD_VP
4 ErnstIT_PROG
5 FayMK_REP
6 GietzAC_ACCOUNT
```

…

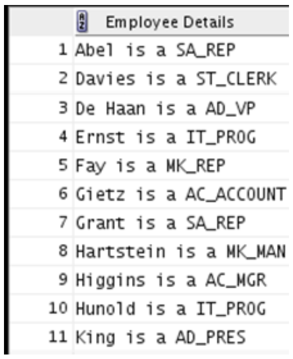
A concatenation operator is represented by two vertical bars (||) and links columns or character strings to other columns. In the above statement, LAST_NAME and JOB_ID are concatenated, and given the alias *Employees*. Note that the last name of the employee and the job code are combined to make a single output column.

Activity 7: Using Literal Character Strings

Write a SQL statement to select last_name and job_id of employees, concatenate them, and display in a single column. Use 'is a' as a separator between last_name and job_id to make it more readable.

```
SELECT     last_name||'is a'||job_id AS"Employee Details"
FROM       employees;
```

Output



```
   Employee Details
 1 Abel is a SA_REP
 2 Davies is a ST_CLERK
 3 De Haan is a AD_VP
 4 Ernst is a IT_PROG
 5 Fay is a MK_REP
 6 Gietz is a AC_ACCOUNT
 7 Grant is a SA_REP
 8 Hartstein is a MK_MAN
 9 Higgins is a AC_MGR
10 Hunold is a IT_PROG
11 King is a AD_PRES
```

…

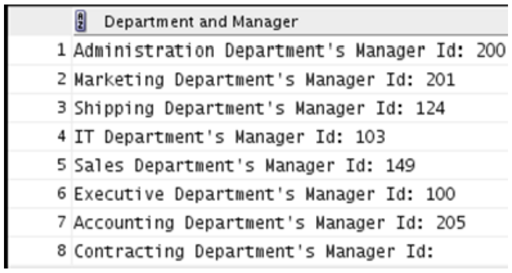
The above statement makes use of a literal to make the column values more readable. A literal is a character, a number, or a date that is included in the SELECT list. It is not a column name or a column alias. It is printed for each row returned. Literal strings of free- format text can be included in the query result and are treated the same as a column in the SELECT list. The date and character literals must be enclosed within single quotation marks (' '); number literals need not be enclosed in a similar manner.

Activity 8: Using Alternative Quote (*q*) Operator

Write a SQL statement to select department_name and manager_id of departments, concatenate them, and display in a single column. Use 'Department's Manager Id:' as a separator between department_name and manager_id.

```
SELECT      department_name || q'[ Department's Manager Id: ]'
            || manager_id AS "Department and Manager"
FROM        departments;
```

Output



```
  Department and Manager
1 Administration Department's Manager Id: 200
2 Marketing Department's Manager Id: 201
3 Shipping Department's Manager Id: 124
4 IT Department's Manager Id: 103
5 Sales Department's Manager Id: 149
6 Executive Department's Manager Id: 100
7 Accounting Department's Manager Id: 205
8 Contracting Department's Manager Id:
```

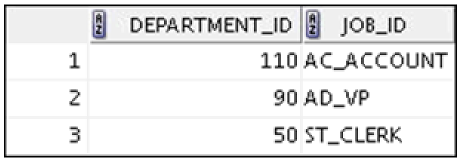
Many SQL statements use character literals in expressions or conditions. If the literal itself contains a single quotation mark, you can use the quote (*q*) operator and select your own quotation mark delimiter. You can choose any convenient delimiter, single-byte or multibyte, or any of the following character pairs: [ ], { }, ( ), or < >. In the above statement, the string contains a single quotation mark, which is normally interpreted as a delimiter of a character string. By using the *q* operator, however, brackets [ ] are used as the quotation mark delimiters. The string between the brackets delimiters is interpreted as a literal character string.

Activity 9: Using DISTINCT Keyword to Get Rid of Duplicates

Write a SQL statement to select only department_id and job_id columns of the employees table. Remove all duplicate rows from the result

```
SELECT      DISTINCT department_id, job_id
FROM        employees;
```

Output



```
   DEPARTMENT_ID JOB_ID
1            110 AC_ACCOUNT
2             90 AD_VP
3             50 ST_CLERK
```

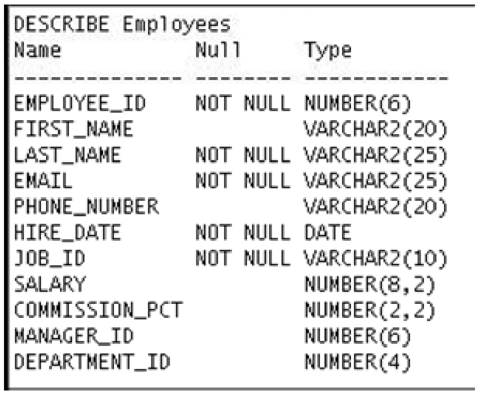
…

By default, SQL displays the results of a query without eliminating the duplicate rows. To eliminate duplicate rows in the result, we include the DISTINCT keyword in the SELECT clause immediately after the SELECT keyword. The DISTINCT qualifier affects all the selected columns, and the result is every distinct combination of the columns.

Activity 10: Using the DESCRIBE Keyword

Display the structure of employees table.

```
DESCRIBE    employees;
```



```
DESCRIBE Employees
Name           Null     Type
-------------- -------- ------------
EMPLOYEE_ID    NOT NULL NUMBER(6)
FIRST_NAME              VARCHAR2(20)
LAST_NAME      NOT NULL VARCHAR2(25)
EMAIL          NOT NULL VARCHAR2(25)
PHONE_NUMBER            VARCHAR2(20)
HIRE_DATE      NOT NULL DATE
JOB_ID         NOT NULL VARCHAR2(10)
SALARY                  NUMBER(8,2)
COMMISSION_PCT          NUMBER(2,2)
MANAGER_ID              NUMBER(6)
DEPARTMENT_ID           NUMBER(4)
```

Output

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Lab Task 1

Write a SQL statement that displays the last_name, job_id, hire_date, and employee_id for each employee, with the employee_id appearing first. Rename HIRE_DATE column as STARTDATE.

Lab Task 2

Write a SQL statement that displays the last_name concatenated with the job_id (separated by a comma and space) of all the employees and name the columns as Employee and Title.

Lab Task 3

Write a SQL statement that instead of retrieving all rows from employees table for all job_id's, find only distinct job_id's.

Lab Task 4

Write a SQL statement that shows first_name and last_name of all employee after concatenation them. Use space as the separator. Name the resultant column as "Name"

Lab Task 5

Write a SQL statement that displays first_name and the annual salary of employees.

Lab Task 6

Write a SQL statement that displays the structure of the departments table.

Lab 03

Restricting and Sorting Data

Objective:

The objective of this lab is to know few other clauses that can be used in SELECT statement to restrict the number of resultant rows (WHERE clause) and to sort the data (ORDER BY clause). Apart from that, various other operators such as BETWEEN, IN, and LIKE that can be used in the WHERE clause to form a condition will also be introduced.

Activity Outcomes:

After completing this lab, students should be able to do the following:

- Use of WHERE clause to limit the number of rows based on some criteria
- Define conditions in WHERE clause using comparison operators
- Use of Character Strings and Dates
- Use of range conditions using the BETWEEN operator
- Membership condition using the IN operator
- Test NULL values
- Pattern matching using the LIKE operator
- Combine conditions in WHERE clause using logical operators
- Sort rows using the ORDER BY clause
- Use of Substitution Variables

Instructor Note:

Please make sure that you have completed the previous labs.

1) Useful Concepts

We can restrict the rows that are returned from the SELECT statement by using the WHERE clause. A WHERE clause contains a condition that must be met and it directly follows the FROM clause. If the condition is true, the row meeting the condition is returned.
Here is the basic form of SELECT statement with optional WHERE clause:

```
SELECT     *|{[DISTINCT] column|expression [alias],...}
FROM       table
[WHERE     logical expression(s)];
```

WHERE	Restricts the query to rows that meet a condition logical expression
logical expression	Is composed of column names, constants, and a comparison operator. It specifies a combination of one or more expressions and Boolean operators, and returns a value of TRUE, FALSE, or UNKNOWN

The WHERE clause can compare values in columns, literal, arithmetic expressions, or functions. Here's the list of comparison operators that can be used in WHERE clause.

Operator	Meaning
=	Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
<>	Not equal to
BETWEEN ... AND ...	Between two values (inclusive)
IN (set)	Match any of a list of values
LIKE	Match a character pattern
IS NULL	Is a null value

Comparison operators are used in conditions that compare one expression with another value or expression. They are used in the WHERE clause in the following format:

WHERE *expr operator value*
e.g.
... WHERE hire_date = '01-JAN-95'
... WHERE salary >= 6000
... WHERE last_name = 'Smith'

Here's the list of logical operators that can be used in WHERE clause:

Operator	Meaning	Use
AND	Returns TRUE if *both* component conditions are true	*condition1* AND *condition2*
OR	Returns TRUE if *either* component condition is true	*condition1* OR *condition2*
NOT	Returns TRUE if the condition is false	*NOT condition*

Since we can use various operators inside WHERE clause, so it's very important to know the precedence of various operators. The rules of precedence determine the order in which expressions are evaluated and calculated. The following table lists the default order of precedence. However, we can override the default order by using parentheses around the expressions that you want to calculate first.
The rules of precedence determine the order in which expressions are evaluated and calculated. The table in the slide lists the default order of precedence. However, you can override the default order by using parentheses around the expressions that you want to calculate first.

Precedence	Operator
1	Arithmetic operators (/, *, +, -)
2	Concatenation operator
3	Comparison conditions
4	IS [NOT] NULL, LIKE, [NOT] IN
5	[NOT] BETWEEN
6	Not equal to
7	NOT logical condition
8	AND logical condition
9	OR logical condition

The SELECT statement can also be used to order the resultant rows. For this, the ORDER BY clause is used in SELECT statement to sort the rows. However, if we use the ORDER BY clause, it must be the last clause of the SQL statement. Further, we can specify an expression, an alias, or a column position as the sort condition. Here's the format of SELECT statement with ORDER BY clause:

```
SELECT      expr
FROM        table
[WHERE      condition(s)]
[ORDER BY   {column, expr, numeric_position} [ASC|DESC]];
```

In the ORDER BY clause, ASC orders the rows in ascending order (This is the default order.) while DESC orders the rows in descending order

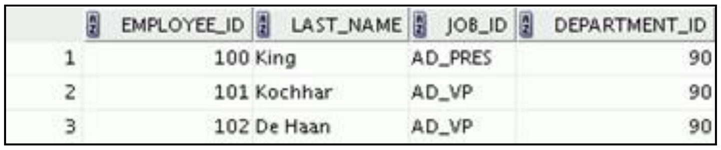
2) Solved Lab Activites

Activity No	*Allocated Time*	*Level of complexity*	*CLO Mapping*
Activity 1	***5 mins***	***Low***	***CLO-5***
Activity 2	***5 mins***	***Low***	***CLO-5***
Activity 3	***5 mins***	***Low***	***CLO-5***
Activity 4	***5 mins***	***Low***	***CLO-5***
Activity 5	***5 mins***	***Low***	***CLO-5***
Activity 6	***5 mins***	***Low***	***CLO-5***
Activity 7	***5 mins***	***Low***	***CLO-5***
Activity 8	***5 mins***	***Low***	***CLO-5***
Activity 9	***10 mins***	***Medium***	***CLO-5***
Activity 10	***10 mins***	***Medium***	***CLO-5***

Activity 1: Using the WHERE Clause

Write a SELECT statement that retrieves the employee_id, last_name, job_id, and department_id of all employees whose department_id is 90.

```
SELECT    employee_id, last_name, job_id, department_id
FROM      employees
WHERE     department_id = 90;
```



```
   EMPLOYEE_ID LAST_NAME JOB_ID  DEPARTMENT_ID
1          100 King      AD_PRES            90
2          101 Kochhar   AD_VP              90
3          102 De Haan   AD_VP              90
```

Output

Activity 2: Using the WHERE Clause with Character Strings and Dates

Write a SELECT statement that retrieves the last_name, job_id, and department_id of all employees whose last name is 'Whalen'

Write another SELECT statement that retrieves the last_name of all employees whose hiring date is '17-FEB-96'

```
SELECT    last_name, job_id, department_id
```

```
FROM      employees
WHERE     last_name = 'Whalen';
```

```
SELECT    last_name
FROM      employees
WHERE     hire_date = '17-FEB-96';
```

Output

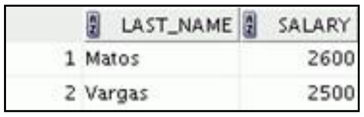
Character strings and date values are enclosed with single quotation marks. Character values are case-sensitive and date values are format-sensitive. The default date display format is DD-MON-YY.

Activity 3: Using Comparison Operators

Write a SELECT statement that retrieves the last name and salary from the EMPLOYEES table for any employee whose salary is less than or equal to 3000

```
SELECT    last_name, salary
FROM      employees
WHERE     salary <= 3000;
```

Output



	LAST_NAME	SALARY
1	Matos	2600
2	Vargas	2500

Activity 4: Range Conditions Using the BETWEEN Operator

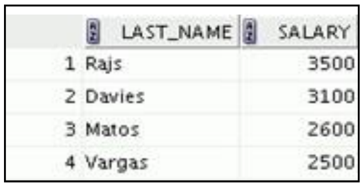
Write a SELECT statement that retrieves the last name and salary from the EMPLOYEES table for any employee whose salary is between 2500 and 3500 (Both inclusive).

Write another SELECT statement that retrieves the last name from the EMPLOYEES table for any employee whose last name is between 'King' AND 'Smith' (Both inclusive).

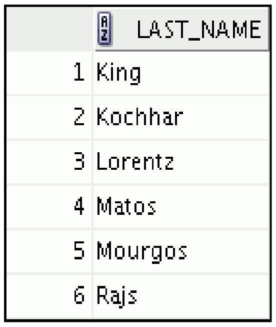
```
SELECT    last_name, salary
FROM      employees
WHERE     salary BETWEEN 2500 AND 3500;
```

```
SELECT    last_name
FROM      employees
WHERE     last_name BETWEEN 'King' AND 'Smith'
```

Output



	LAST_NAME	SALARY
1	Rajs	3500
2	Davies	3100
3	Matos	2600
4	Vargas	2500



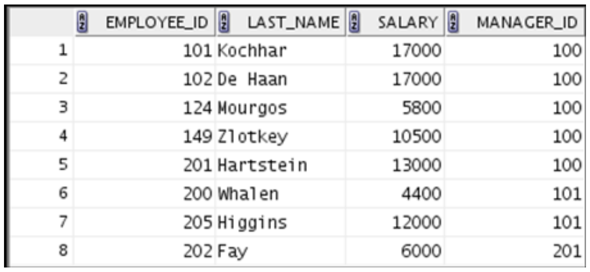
	LAST_NAME
1	King
2	Kochhar
3	Lorentz
4	Matos
5	Mourgos
6	Rajs

Activity 5: Membership Condition Using the IN Operator

Write a SELECT statement that retrieves last_name, salary, and manager_id for all the employees who's last_name is either 'Hartstein' or 'Vargas'

```
SELECT    last_name, salary, manager_id
FROM      employees
WHERE     last_name IN ('Hartstein', 'Vargas');
```

Output



	EMPLOYEE_ID	LAST_NAME	SALARY	MANAGER_ID
1	101	Kochhar	17000	100
2	102	De Haan	17000	100
3	124	Mourgos	5800	100
4	149	Zlotkey	10500	100
5	201	Hartstein	13000	100
6	200	Whalen	4400	101
7	205	Higgins	12000	101
8	202	Fay	6000	201

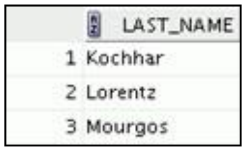
IN operator is used to test for values in a specified set of values. The condition defined using the IN operator is also known as the membership condition. The set of values can be specified in any random order. The IN operator can be used with any data type.

Activity 6: Pattern Matching Using the LIKE Operator

Write another SELECT statement that retrieves the last name of all employees whose last names have the letter "o" as the second character.

```
SELECT     last_name
FROM       employees
WHERE      last_name LIKE '_o%'
```

Output



	LAST_NAME
1	Kochhar
2	Lorentz
3	Mourgos

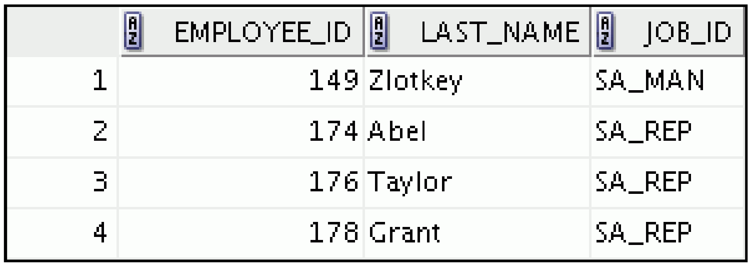
LIKE operator is very useful when we may not know the exact value to search for. In such situations we can select rows that match a character pattern by using the LIKEoperator. The character pattern–matching operation is referred to as a *wildcard* search. Two symbols can be used to construct the search string:

- '%' which denotes zero or many characters.
- '_' (underscore) which denotes one character

There could be cases when we actually need to have an exact match for the actual %and _characters. In this case, we can use the ESCAPE identifier. This option specifies what the escape character is. If you want to search for strings that contain SA_, you can use the following SQL statement:

```
SELECT     employee_id, last_name, job_id
FROM       employees
WHERE      job_id LIKE '%SA\_%' ESCAPE '\';
```

In the above SELECT statement, the ESCAPE identifier identifies the backslash (\) as the escape character. In the SQL statement, the escape character precedes the underscore (_). This causes the Oracle server to interpret the underscore literally.



	EMPLOYEE_ID	LAST_NAME	JOB_ID
1	149	Zlotkey	SA_MAN
2	174	Abel	SA_REP
3	176	Taylor	SA_REP
4	178	Grant	SA_REP

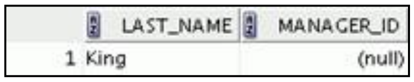
Activity 7: Testing for NULL values

Write a SELECT statement that retrieves the last names and managers of all employees who do not have a manager.

```
SELECT     last_name, manager_id
FROM       employees
```

```
WHERE       manager_id IS NULL;
```

Output



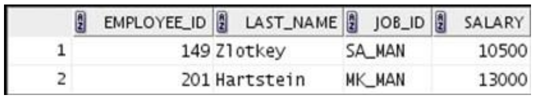
	LAST_NAME	MANAGER_ID
1	King	(null)

The NULL conditions include the IS NULL condition and the IS NOT NULL condition. The IS NULL condition tests for nulls. A null value means that the value is unavailable, unassigned, unknown, or inapplicable. Therefore, you cannot test with =, because a null cannot be equal or unequal to any value

Activity 8: Combining Conditions using Logical Operators

Write a SELECT statement that retrieves the employee_id, last_name, job_id, salary of all employees who have a job_id that contains the string 'MAN' and earn 10,000

```
SELECT      employee_id, last_name, job_id, salary
FROM        employees
WHERE       salary >= 10000 AND job_id LIKE '%MAN%';
```



	EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
1	149	Zlotkey	SA_MAN	10500
2	201	Hartstein	MK_MAN	13000

Output

Activity 9: Sorting Rows using the ORDER BY Clause

Write a SELECT statement that retrieves the last_name, job_id, department_id, hire_date of all employees. Sort the result by hire_date in ascending order.

Write another SELECT retrieves the last_name, job_id, department_id, hire_date of all employees. This time, sort the result by hire_date in descending order.

Write another SELECT retrieves the last_name, job_id, department_id, hire_date of all employees. This time, sort the result by the third column in ascending order.

Write another SELECT retrieves the last_name, job_id, department_id, hire_date of all employees. This time, first sort the result by the department_id in ascending order and then by hire_date in descending order

```
SELECT      last_name, job_id, department_id, hire_date
FROM        employees
```

```
ORDER BY    hire_date;
```

```
SELECT      last_name, job_id, department_id, hire_date
FROM        employees
ORDER BY    hire_date DESC;
```

```
SELECT      last_name, job_id, department_id, hire_date
FROM        employees
ORDER BY    3;
```

```
SELECT      last_name, job_id, department_id, hire_date
FROM        employees
ORDER BY    department_id, hire_date DESC;
```

Output

If the ORDER BY clause is not used, the sort order is undefined, and the Oracle server may not fetch rows in the same order for the same query twice. We can sort query results by specifying the numeric position of the column in the SELECT clause as done in the 3rd query above. We can also sort query results by more than one column. The sort limit is the number of columns in the given table. In the ORDER BY clause, we can specify the columns and separate the column names using commas as done in the 4th query.

Activity 10: Using the single-ampersand and double ampersand substitution variables

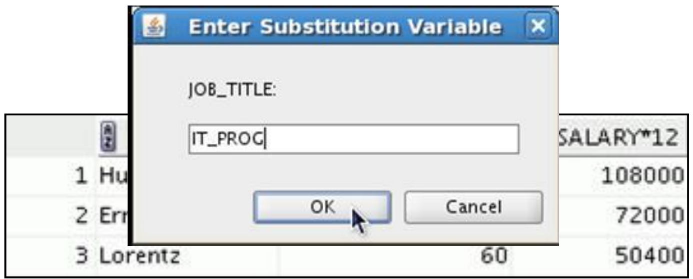
Write a SELECT statement that retrieves the last_name, department_id, and annual salary of all employees whose job_id is provided by the user.

Write another SELECT statement that retrieves the employee_id, last_name, job_id, and another column of user's choice of all employees. Sort the rows too by the user provided column name.

```
SELECT      last_name, department_id, salary*12
FROM        employees
WHERE       job_id = '&job_title';
```

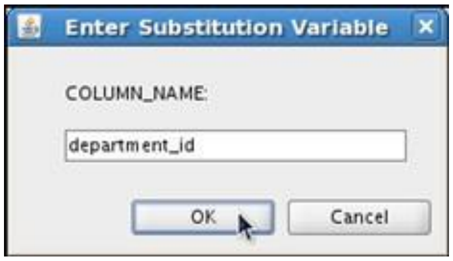
```
SELECT      employee_id, last_name, job_id, && columnname
FROM        employees
ORDER BY    &columnname;
```

Output

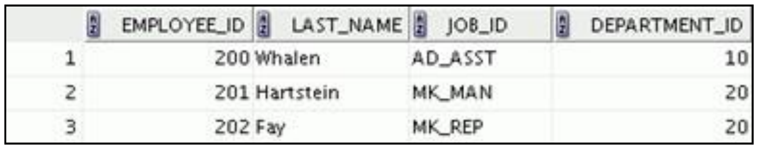


JOB_TITLE: IT_PROG

```
                          SALARY*12
  1 Hu                       108000
  2 Err                       72000
  3 Lorentz          60       50400
```



COLUMN_NAME: department_id

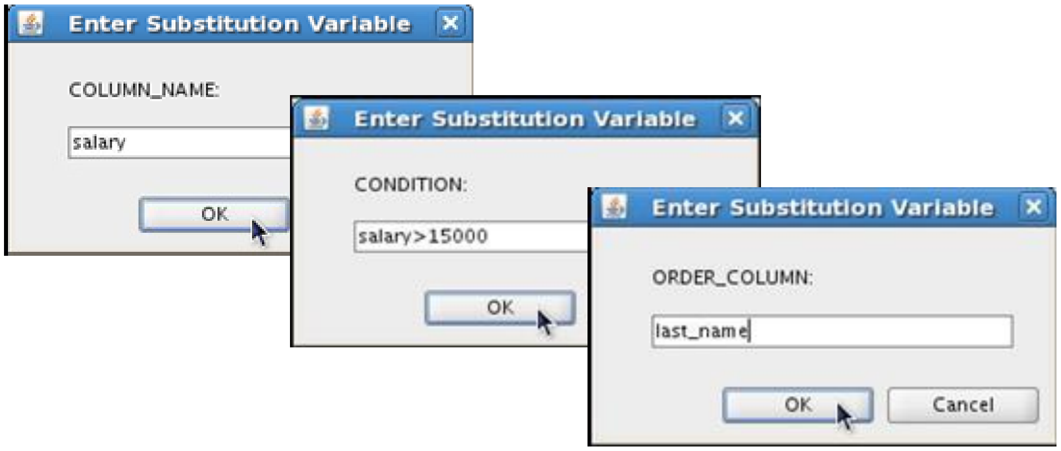


	EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
1	200	Whalen	AD_ASST	10
2	201	Hartstein	MK_MAN	20
3	202	Fay	MK_REP	20

…

When running a query, users often want to dynamically specify the variable based on which the WHERE clause would restrict the data. SQL Developer provides this flexibility with user variables. We can use an ampersand (&) to identify each such variable in our SQL statement. In the first query, we have created a SQL Developer substitution variable for the *job_id*. When the statement is executed, SQL Developer prompts the user for a *job_id* (Variable name is set to job_title) and then displays the last name, department number and the annual salary for that employee.
With the single ampersand, the user is prompted every time the variable is used. We can use the double-ampersand (&&) substitution variable if we want to reuse the variable value without prompting the user each time. The user sees the prompt for the value only once. In the 2nd query, the user is asked to give the value for the variable, column_name, only once. The value that is supplied by the user (department_id) is used for both display and ordering of data. If you run the query again, you will not be prompted for the value of the variable.
We can use the substitution variables not only in the WHERE clause of a SQL statement, but also as substitution for column names, expressions, or text as demonstrated in the following statement:

```
SELECT      employee_id,last_name,job_id,&columnname
FROM        employees
WHERE       &condition
ORDER BY    &order_column;
```


Enter Substitution Variable
OK
Enter Substitution Variable
OK
Enter Substitution Variable
OK
Cancel

COLUMN_NAME: salary

CONDITION: salary>15000

ORDER_COLUMN: last_name

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Lab Task 1

Write a SQL statement that retrieves the last name and department number for employee number 176.

Lab Task 2

Write a SQL statement that retrieves the last name and salary for any employee whose salary is not in the range of 5,000 to 12,000.

Lab Task 3

Write a SQL statement that retrieves the last name, job ID, and hire date for employees with the last names of Matos or Taylor. Sort the resultant rows in ascending order by the hire date.

Lab Task 4

Write a SQL statement that retrieves the last name and department ID of all employees in departments 20 or 50 in ascending alphabetical order by name.

Lab Task 5

Write a SQL statement that retrieves the last name and job title of all employees whose department ID is unknown.

Lab Task 6

Write a SQL statement that retrieves the last name, salary, and commission of all employees who earn commissions (i.e., the commission_pct is not NULL). Sort the resultant rows in descending order of salary and then commissions. Use the salary column's numeric position in the ORDER BY clause..

Lab Task 7

Write a SQL statement that prompts the user for a manager ID and retrieves the employee ID, last name, salary, and department for that manager's employees. The query must also prompt the column based on which the resultant rows will be sorted in ascending order. Test the data with the following values:

- *manager_id=103, sorted by last_name*
- *manager_id = 201, sorted by salary*
- *manager_id = 124, sorted by employee_id*

Lab Task 8

Write a SQL statement that retrieves the last names of all employees who have both an "a" and an "e" somewhere in any order in their last name.

Lab Task 9

Write a SQL statement that retrieves the last name, job, and salary for all employees whose jobs are either those of a sales representative or of a stock clerk, and whose salaries are not equal to 2500, 3500, or 7000.

Lab 04

Using Single Row Functions to Customize Output

Objective:

The objective of this lab is to introduce various single rows functions available in SQL to customize the output.

Activity Outcomes:

After completing this lab, students should be able to do the following:

- Describe the various types of functions available in SQL
- Use the character, number, and date functions in SELECT statements

Instructor Note:

Please make sure that you have completed the previous labs.

1) Useful Concepts

Single-row functions are used to manipulate data items. They accept one or more arguments and return one value for each row that is returned by the query. An argument can be one of the following:

- User-supplied constant
- Variable value
- Column name
- Expression

Features of single-row functions include:

- Acting on each row that is returned in the query
- Returning one result per row
- Possibly returning a data value of a different type than the one that is referenced
- Possibly expecting one or more arguments
- Can be used in SELECT, WHERE, and ORDER BY clauses; can be nested.

```
function_name [(arg1, arg2,...)]
```

In the above syntax:

- *function_name* is the name of the function
- *arg1*, *arg2* are any argument to be used by the function. This can be represented by a column name or expression.

This lab covers the following single-row functions:

- **Character functions:** Accept character input and can return both character and number values

Function	Purpose
LENGTH(*column*|*expression*)	Returns the number of characters in the expression
INSTR(*column*|*expression*, *'string', [,m], [n]*)	Returns the numeric position of a named string. Optionally, you can provide a position *m* to start searching, and the occurrence *n* of the string. *m* and *n* default to 1, meaning start the search at the beginning of the string and report the first occurrence.

LPAD(*column*|*expression*, *n*, '*string*')
RPAD(*column*|*expression*, *n*, '*string*')
	Returns an expression left-padded to length of *n* characters with a character expression.
	Returns an expression right-padded to length of *n* characters with a character expression.

TRIM(*leading*|*trailing*|*both*, *trim_character FROM trim_source*)	Enables you to trim leading or trailing characters (or both) from a character string. If *trim_character* or *trim_source* is a character literal, you must enclose it in single quotation marks. This is a feature that is available in Oracle8*i* and later versions.
REPLACE(*text*, *search_string*, *replacement_string*)	Searches a text expression for a character string and, if found, replaces it with a specified replacement string

- **Number functions:** Accept numeric input and return numeric values

Function	Result
ROUND(45.926, 2)	45.93
TRUNC(45.926, 2)	45.92
MOD(1600, 300)	100

- **Date functions:** Operate on values of the DATEdata type (All date functions return a value of the DATEdata type except the MONTHS_BETWEENfunction, which returns a number.)

Function	Result
MONTHS_BETWEEN	Number of months between two dates
ADD_MONTHS	Add calendar months to date
NEXT_DAY	Next day of the date specified
LAST_DAY	Last day of the month
ROUND	Round date
TRUNC	Truncate date

The Oracle Database stores dates in an internal numeric format, representing the century, year, month, day, hours, minutes, and seconds. The default display and input format for any date is DD-MON-RR. Valid Oracle dates are between January 1, 4712 B.C., and December 31, 9999 A.D. Because the database stores dates as numbers, you can perform calculations using arithmetic operators such as addition and subtraction. You can add and subtract number constants as well as dates.

The following operations can be performed over dates:

Operation	Result	Description
date + number	Date	Adds a number of days to a date
date – number	Date	Subtracts a number of days from a date
date – date	Number of days	Subtracts one date from another
date + number/24	Date	Adds a number of hours to a date

- **Conversion functions:** Convert a value from one data type to another

Function	Purpose
TO_CHAR(*number*|*date*,[*fmt*], [*nlsparams*])	Date conversion: The nlsparams parameter specifies the language in which the month and day names, and abbreviations are returned. If this parameter is omitted, this function uses the default date languages for the session.

TO_NUMBER(*char,[fmt], [nlsparams]*)	Converts a character string containing digits to a number in the format specified by the optional format model *fmt*. The nlsparams parameter has the same purpose in this function as in the TO_CHAR function for number conversion.
TO_DATE(*char*,[*fmt*],[*nlspara ms*])	Converts a character string representing a date to a date value according to *fmt* that is specified. If *fmt* is omitted, the format is DD-MON-YY. The nlsparams parameter has the same purpose in this function as in the TO_CHAR function for date conversion.

- **General functions:**

Function	Description
NVL	Converts a null value to an actual value
NVL2	If expr1 is not null, NVL2 returns expr2. If expr1 is null, NVL2 returns expr3. The argument expr1can have any data type.
NULLIF	Compares two expressions and returns null if they are equal; returns the first expression if they are not equal
COALESCE	Returns the first non-null expression in the expression list

2) Solved Lab Activites

Activity No	*Allocated Time*	*Level of complexity*	*CLO Mapping*
Activity 1	***10 mins***	***Medium***	***CLO-5***
Activity 2	***10 mins***	***Medium***	***CLO-5***
Activity 3	***10 mins***	***Medium***	***CLO-5***
Activity 4	***10 mins***	***Medium***	***CLO-5***
Activity 5	***10 mins***	***Medium***	***CLO-5***
Activity 6	***10 mins***	***Medium***	***CLO-5***

Activity 1: Use of Character Functions to Display Well-Formatted Results

In this lab activity, we are going to perform the following three tasks:

- ***Display the following information for all of the employees in the given format:***
  The job id for 'last_name' is job_id
  where last_name must be in upper case and job_id in lower case.

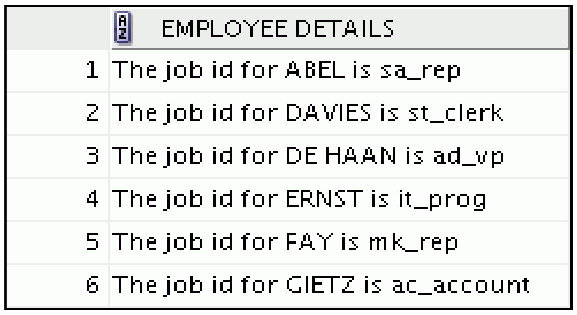
- ***Display the employee number, name, and department number for employee Higgins.***
- ***Display employee first names and last names joined together, the length of the employee's last name, and the numeric position of the letter "a" in the employee's last name for all employees who have the string, REP, contained in the job ID starting at the fourth position of the job ID.***

```
SELECT   'The job id for '||UPPER(last_name)||' is '||LOWER(job_id) AS
         "EMPLOYEE DETAILS"
FROM     employees
```

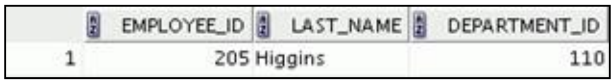
```
SELECT   employee_id,last_name,department_id
FROM     employees
WHERE    INITCAP(last_name) = 'Higgins'
```

```
SELECT   employee_id,CONCAT(first_name, last_name) NAME,job_id,
         LENGTH(last_name),INSTR(last_name, 'a') "Contains 'a'?"
FROM     employees
WHERE    SUBSTR(job_id, 4) = 'REP';
```

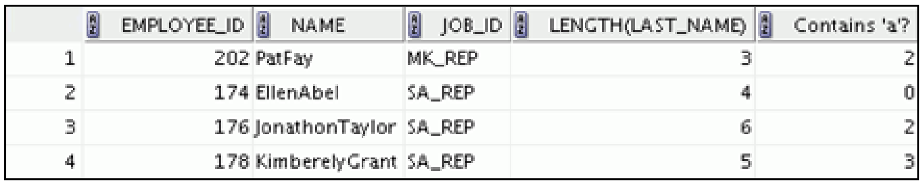
Output



	EMPLOYEE DETAILS
1	The job id for ABEL is sa_rep
2	The job id for DAVIES is st_clerk
3	The job id for DE HAAN is ad_vp
4	The job id for ERNST is it_prog
5	The job id for FAY is mk_rep
6	The job id for GIETZ is ac_account



	EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
1	205	Higgins	110



	EMPLOYEE_ID	NAME	JOB_ID	LENGTH(LAST_NAME)	Contains 'a'?
1	202	PatFay	MK_REP	3	2
2	174	EllenAbel	SA_REP	4	0
3	176	JonathonTaylor	SA_REP	6	2
4	178	KimberelyGrant	SA_REP	5	3

Activity 2: Use of Number Functions

In this lab activity, we are going to perform the following two tasks:

- ***For all employees with the job title of Sales Representative, calculate the remainder of the salary after it is divided by 5,000.***

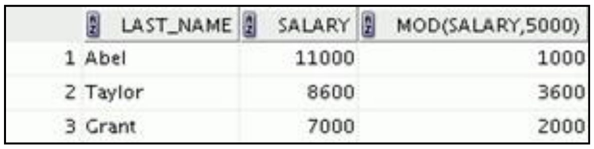
- ***Check how truncation and rounding works***

```
SELECT   last_name, salary, MOD(salary, 5000) FROM employees
WHERE    job_id = 'SA_REP';
```

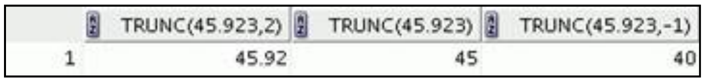
```
SELECT   TRUNC(45.923,2),TRUNC(45.923,TRUNC(45.923,-1)
FROM     DUAL;
```

```
SELECT   ROUND(45.923,2), ROUND(45.923, 0), ROUND(45.923,-1)
FROM     DUAL;
```

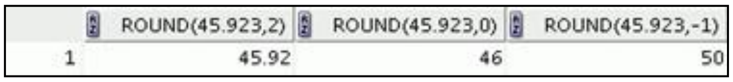
Output



	LAST_NAME	SALARY	MOD(SALARY,5000)
1	Abel	11000	1000
2	Taylor	8600	3600
3	Grant	7000	2000



	TRUNC(45.923,2)	TRUNC(45.923)	TRUNC(45.923,-1)
1	45.92	45	40



	ROUND(45.923,2)	ROUND(45.923,0)	ROUND(45.923,-1)
1	45.92	46	50

Note: The DUALtable used in above examples is owned by the user SYSand can be accessed by all users. It contains one column, DUMMY, and one row with the value X.

Activity 3: Use of Date Functions

In this lab activity, we are going to perform the following three tasks:

- ***Display the last name and the number of weeks employed for all employees in department 90.***
- ***Display the employee number, hire date, number of months employed, six- month review date, first Friday after hire date, and the last day of the hire month for all employees who have been employed for fewer than 150 months***
- ***For all employees who started in 1997, display the employee number, hire date, and starting month using the ROUND and TRUNC functions.***

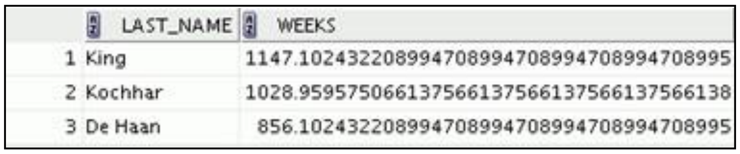
```
SELECT   last_name, (SYSDATE-hire_date)/7 AS WEEKS
FROM     employees
WHERE    department_id = 90;
```

```
SELECT   employee_id,hire_date,MONTHS_BETWEEN (SYSDATE,hire_date) TENURE,
         ADD_MONTHS (hire_date, 6) REVIEW,NEXT_DAY (hire_date, 'FRIDAY'),
         LAST_DAY(hire_date)
```

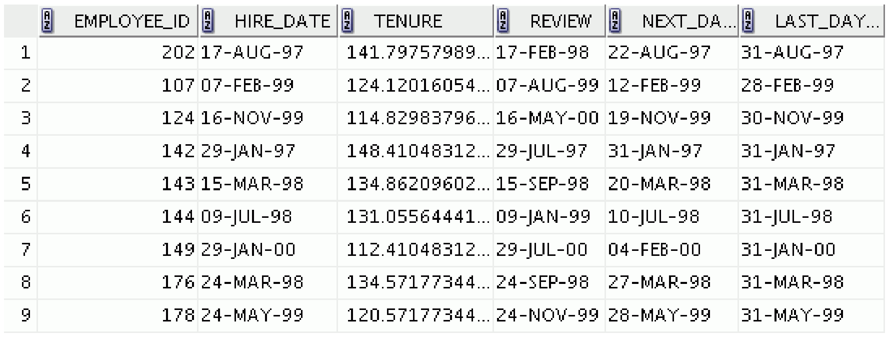
```
FROM      employees
WHERE     MONTHS_BETWEEN (SYSDATE, hire_date) < 150;
```

```
SELECT    employee_id, hire_date, ROUND(hire_date,'MONTH'),
          TRUNC(hire_date, 'MONTH')
FROM      employees
WHERE     hire_date LIKE '%97'
```

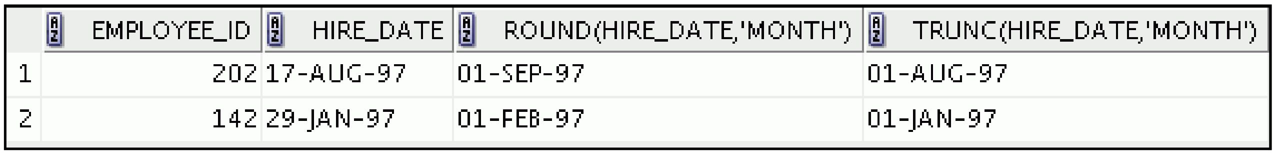
Output



	LAST_NAME	WEEKS
1	King	1147.102432208994708994708994708994708995
2	Kochhar	1028.959575066137566137566137566137566138
3	De Haan	856.102432208994708994708994708994708995



	EMPLOYEE_ID	HIRE_DATE	TENURE	REVIEW	NEXT_DA...	LAST_DAY...
1	202	17-AUG-97	141.79757989...	17-FEB-98	22-AUG-97	31-AUG-97
2	107	07-FEB-99	124.12016054...	07-AUG-99	12-FEB-99	28-FEB-99
3	124	16-NOV-99	114.82983796...	16-MAY-00	19-NOV-99	30-NOV-99
4	142	29-JAN-97	148.41048312...	29-JUL-97	31-JAN-97	31-JAN-97
5	143	15-MAR-98	134.86209602...	15-SEP-98	20-MAR-98	31-MAR-98
6	144	09-JUL-98	131.05564441...	09-JAN-99	10-JUL-98	31-JUL-98
7	149	29-JAN-00	112.41048312...	29-JUL-00	04-FEB-00	31-JAN-00
8	176	24-MAR-98	134.57177344...	24-SEP-98	27-MAR-98	31-MAR-98
9	178	24-MAY-99	120.57177344...	24-NOV-99	28-MAY-99	31-MAY-99



	EMPLOYEE_ID	HIRE_DATE	ROUND(HIRE_DATE,'MONTH')	TRUNC(HIRE_DATE,'MONTH')
1	202	17-AUG-97	01-SEP-97	01-AUG-97
2	142	29-JAN-97	01-FEB-97	01-JAN-97

Note: SYSDATE is a date function that returns the current database server date and time.

Activity 4: Using Conversion Functions and Conditional Expressions

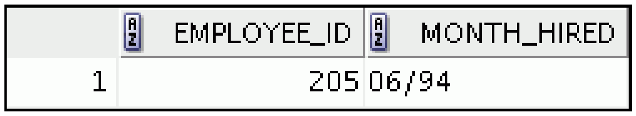
In this lab activity, we are going to perform the following three tasks:

- ***Use TO_CHAR to convert a datetime data type to a value of VARCHAR2 data type***
- ***Display employees hired before 1990, using the RR date format, which produces the same results whether the command is run in 1999 or now***

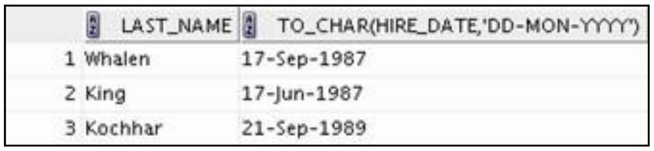
```
SELECT    employee_id, TO_CHAR(hire_date,  'MM/YY')  Month_Hired
FROM      employees
WHERE     last_name = 'Higgins';
```

```
SELECT   last_name, TO_CHAR(hire_date, 'DD-Mon-YYYY')
FROM     employees
WHERE    hire_date < TO_DATE('01-Jan-90','DD-Mon-RR');
```

Output



	EMPLOYEE_ID	MONTH_HIRED
1	205	06/94



	LAST_NAME	TO_CHAR(HIRE_DATE,'DD-MON-YYYY')
1	Whalen	17-Sep-1987
2	King	17-Jun-1987
3	Kochhar	21-Sep-1989

Activity 5: Nesting Functions

Single-row functions can be nested to any depth. Nested functions are evaluated from the innermost level to the outermost level. In this activity, we are going to perform the following two tasks related to nested functions:

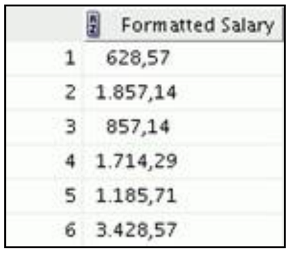
- ***Display the date of the next Friday that is six months from the hire date. The resulting date should appear as Friday, August 13th, 1999. Order the results by hire date.***
- ***Display the salaries of employees divided by 7 and rounded to two decimals. Display the salary in Danish notation. That is, comma is used for decimal point and a period for thousands.***

```
SELECT    TO_CHAR(NEXT_DAY(ADD_MONTHS(hire_date, 6), 'FRIDAY'), 'fmDay,
          Month ddth, YYYY') "Next 6 Month Review"
FROM      employees
ORDER BY  hire_date;
```

```
SELECT    TO_CHAR(ROUND((salary/7), 2),'99G999D99',
          'NLS_NUMERIC_CHARACTERS = '',.'' ') "Formatted Salary"
FROM      employees
```

Output

The output for the 2nd query is:



	Formatted Salary
1	628,57
2	1.857,14
3	857,14
4	1.714,29
5	1.185,71
6	3.428,57

…

Activity 6: Using General Functions

In this lab activity, we are going to perform the following three tasks:

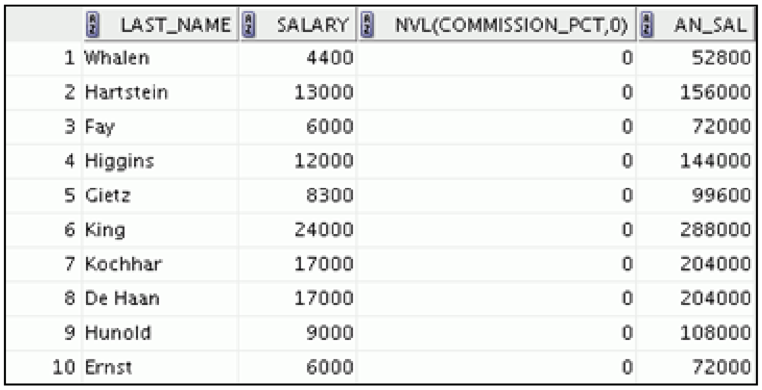
- ***Calculate the annual compensation of all employees including those with no commission percentage i.e., commission_pct is NULL.***
- ***Demonstrate the use of NVL2 function by examining the COMMISSION_PCT column. If a value is detected, the text literal value of SAL+COMM is returned. If the COMMISSION_PCT column contains a null value, the text literal value of SAL is returned.***
- ***Demonstrate the use of NULLIF function by comparing the length of the first name in the EMPLOYEES table to the length of the last name in the EMPLOYEES table.***

```
SELECT      last_name, salary, NVL(commission_pct, 0),
            (salary*12) + (salary*12*NVL(commission_pct, 0)) AN_SAL
FROM        employees;
```

```
SELECT      last_name, salary, NVL2(commission_pct, 'SAL+COMM','SAL')
            income
FROM        employees
WHERE       department_id IN (50, 80);
```

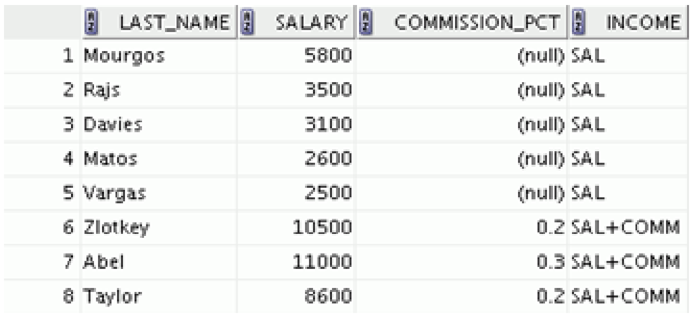
```
SELECT      first_name, LENGTH(first_name) "expr1", last_name,
            LENGTH(last_name)"expr2", NULLIF(LENGTH(first_name),
            LENGTH(last_name)) result
FROM        employees;
```

Output

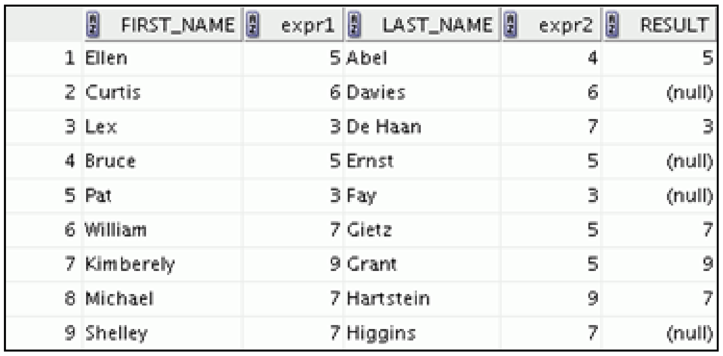


	LAST_NAME	SALARY	NVL(COMMISSION_PCT,0)	AN_SAL
1	Whalen	4400	0	52800
2	Hartstein	13000	0	156000
3	Fay	6000	0	72000
4	Higgins	12000	0	144000
5	Gietz	8300	0	99600
6	King	24000	0	288000
7	Kochhar	17000	0	204000
8	De Haan	17000	0	204000
9	Hunold	9000	0	108000
10	Ernst	6000	0	72000

…



	LAST_NAME	SALARY	COMMISSION_PCT	INCOME
1	Mourgos	5800	(null)	SAL
2	Rajs	3500	(null)	SAL
3	Davies	3100	(null)	SAL
4	Matos	2600	(null)	SAL
5	Vargas	2500	(null)	SAL
6	Zlotkey	10500	0.2	SAL+COMM
7	Abel	11000	0.3	SAL+COMM
8	Taylor	8600	0.2	SAL+COMM



	FIRST_NAME	expr1	LAST_NAME	expr2	RESULT
1	Ellen	5	Abel	4	5
2	Curtis	6	Davies	6	(null)
3	Lex	3	De Haan	7	3
4	Bruce	5	Ernst	5	(null)
5	Pat	3	Fay	3	(null)
6	William	7	Gietz	5	7
7	Kimberely	9	Grant	5	9
8	Michael	7	Hartstein	9	7
9	Shelley	7	Higgins	7	(null)

…

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Lab Task 1

The HR department needs a report to display the employee number, last name, salary, and salary increased by 15.5% (expressed as a whole number) for each employee. Write a SQL statement for ths need. Label the last column as New Salary.

Lab Task 2

Write a SQL statement that displays the last name (with the first letter in uppercase and all the other letters in lowercase) and the length of the last name for all employees whose name starts with the letters “J”, “A”, or “M”. Give each column an appropriate label. Sort the results by the employees ‘last names. Rewrite the query so that the user is prompted to enter a letter that the last name starts with. For example, if the user enters H (capitalized) when prompted for a letter, then the output should show all employees whose last name starts with the letter H.

Lab Task 3

The HR department wants to find the duration of employment for each employee. Write a SQL statement that for each employee, display the last name and calculate the number of months between today and the date on which the employee was hired. Label the column as MONTHS_WORKED. Order your results by the number of months employed. Round the number of months up to the closest whole number.

Lab Task 4

Write a SQL statement that displays the employee’s last names and commission amounts. If an employee does not earn commission, show — No Commission. Label the column as COMM.

Lab 05

Reporting Aggregated Data Using the Group Functions

Objective:

This lab focuses on obtaining summary information (such as averages) for groups of rows. The objective is to group rows in a table into smaller sets, apply aggregate functions to each group, and to filter the groups.

Activity Outcomes:

After completing this lab, students should be able to:

- Understand the use of grouping and aggregation in generating useful reports from the data in the form of tables
- Identify the available aggregate functions
- Describe the use of aggregate functions COUNT, MAX, MIN, SUM, AVG, STDDEV, VARIANCE
- Write queries to group data by using the GROUP BY clause
- Write queries to include or exclude grouped rows by using the HAVING clause

Instructor Note:

Please make sure that you have completed the previous labs.

1) Useful Concepts

Unlike single-row functions, group functions operate on sets of rows to give one result per group. These sets may comprise the entire table or the table split into groups.

```
SELECT     column, group_function
FROM       table
[WHERE     condition]
[GROUP BY  group_by_expression]
[HAVING    group_condition]
[ORDER BY  column];
```

We can use the GROUP BY clause to divide the rows in a table into groups. We can then use the group functions to return summary information for each group.
In the above syntax, *group_by_expression*, specifies the columns whose values determine the basis for grouping rows.

Guidelines for using GROUP BY clause:

- If you include a group function in a SELECT clause, you cannot select individual column as well, unless the individual column appears in the GROUP BY clause. You receive an error message if you fail to include the column list in the GROUP BY clause.
- Using a WHERE clause, you can exclude rows before dividing them into groups. You can substitute column by an Expression in the SELECT statement.
- You must include the columns in the GROUP BY clause.
- You cannot use a column alias in the GROUP BY clause.

Once the groups have been formed, various aggregate functions can be applied to them. The group function is placed after the SELECT keyword. You may have multiple group functions separated by commas. Each of the group functions accepts an argument. The following table identifies the options that you can use in the syntax:

Function	Description
AVG([DISTINCT|ALL]*n*)	Average value of *n*, ignoring null values
COUNT([DISTINCT|ALL]*expr*)	Number of rows, where *expr* evaluates to something other than null (count all selected rows using *, including duplicates and rows with nulls)
MAX([DISTINCT|ALL]expr)	Maximum value of *expr*, ignoring null values
MIN([DISTINCT|ALL]*expr*)	Minimum value of *expr*, ignoring null values
STDDEV([DISTINCT|ALL]*n*)	Standard deviation of *n*, ignoring null values
SUM([DISTINCT|ALL]*n*)	Sum values of *n*, ignoring null values
VARIANCE([DISTINCT|ALL]*n*)	Variance of *n*, ignoring null values

Further guidelines for using the group functions are following:

- DISTINCT makes the function consider only nonduplicate values; ALL makes it consider every value, including duplicates. The default is ALL and, therefore, does not need to be specified.
- The data types for the functions with an expr argument may be CHAR, VARCHAR2, NUMBER, or DATE.
- All group functions ignore null values. To substitute a value for null values, use the NVL, NVL2, COALESCE, CASE, or DECODE functions.

Among group functions, only the COUNT function has three formats:

- COUNT(*)
- COUNT(expr)
- COUNT(DISTINCT expr)

COUNT(*) returns the number of rows in a table that satisfy the criteria of the SELECT statement, including duplicate rows and rows containing null values in any of the columns. If a WHERE clause is included in the SELECT statement, COUNT(*) returns the number of rows that satisfy the condition in the WHERE clause.

In contrast, COUNT(expr) returns the number of non-null values that are in the column identified by expr. COUNT(DISTINCT expr) returns the number of unique, non-null values that are in the column identified by expr.

We use the HAVING clause to specify the groups that are to be displayed, thus further restricting the groups on the basis of aggregate information. In the syntax, *group_condition* restricts the groups of rows returned to those groups for which the specified condition is true. The Oracle server performs the following steps when you use the HAVING clause:

1. Rows are grouped.
2. The group function is applied to the group.
3. The groups that match the criteria in the HAVING clause are displayed. The HAVING clause can precede the GROUP BY clause, but it is recommended that you place the GROUP BY clause first because it is more logical. Groups are formed and group functions are calculated before the HAVING clause is applied to the groups in the SELECT list.

Note: The WHERE clause restricts rows, whereas the HAVING clause restricts groups.

2) Solved Lab Activites

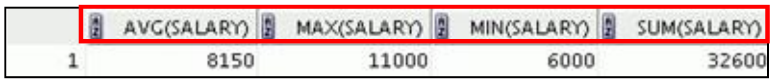
Activity No	***Allocated Time***	***Level of complexity***	***CLO Mapping***
Activity 1	***5 mins***	***Low***	***CLO-5***
Activity 2	***5 mins***	***Low***	***CLO-5***
Activity 3	***5 mins***	***Low***	***CLO-5***
Activity 4	***5 mins***	***Low***	***CLO-5***
Activity 5	***5 mins***	***Low***	***CLO-5***
Activity 6	***10 mins***	***Medium***	***CLO-5***
Activity 7	***5 mins***	***Low***	***CLO-5***
Activity 8	***10 mins***	***Medium***	***CLO-5***
Activity 9	***5 mins***	***Low***	***CLO-5***

Activity 1: Using Aggregate Functions in SELECT statement

Write a SQL statement that displays the average, highest, lowest, and sum of monthly salaries for all sales representatives.

```
SELECT   AVG(salary), MAX(salary), MIN(salary), SUM(salary)
FROM     employees
WHERE    job_id LIKE '%REP%';
```

Output



	AVG(SALARY)	MAX(SALARY)	MIN(SALARY)	SUM(SALARY)
1	8150	11000	6000	32600

You can use the MAX and MIN functions for numeric, character, and date data types. The AVG, SUM, VARIANCE, and STDDEV functions can be used only with numeric data types. MAX and MIN cannot be used with LOB or LONG data types.

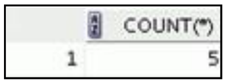
Activity 2: Using the COUNT Function

In this activity, we are first going to use COUNT function to display the number of employees in department 50. In the next statement, we are going to display the number of employees in department 50 who can earn a commission.

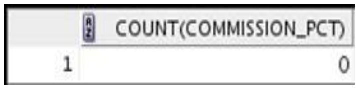
```
SELECT   COUNT(*)
FROM     employees
WHERE    department_id = 50;
```

```
SELECT   COUNT(commission_pct)
FROM     employees
WHERE    department_id = 50;
```

Output



	COUNT(*)
1	5



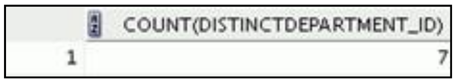
	COUNT(COMMISSION_PCT)
1	0

In the above statements, COUNT(*) returns the number of rows in a table while COUNT(commission_pct) returns the number of rows with non-null values for commission_pct

Activity 3: Using the DISTINCT Keyword

In this activity, we are going to use COUNT with DISTINCT to display the number of distinct department values in the EMPLOYEES table.

```
SELECT   COUNT(DISTINCT department_id)
FROM     employees;
```



```
   COUNT(DISTINCTDEPARTMENT_ID)
1                             7
```

Output

In the above statement, COUNT(DISTINCT department_id) returns the number of distinct non-null values of department_id.

Activity 4: Group Functions and Null Values

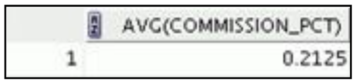
All group functions ignore null values in the column. However, the NVL function forces group functions to include null values. In the activity, we are going to write the two statements to illustrate them.

- ***In the first one, the average is calculated based on only those rows in the table in which a valid value is stored in the COMMISSION_PCT column. The average is calculated as the total commission that is paid to all employees divided by the number of employees receiving a commission.***
- ***In the second one, the average is calculated based on all rows in the table, regardless of whether null values are stored in the COMMISSION_PCT column. The average is calculated as the total commission that is paid to all employees divided by the total number of employees in the company.***

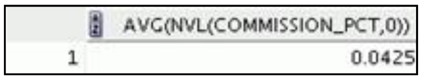
```
SELECT   AVG(commission_pct)
FROM     employees;
```

```
SELECT   AVG(NVL(commission_pct, 0))
FROM     employees;
```

Output



```
   AVG(COMMISSION_PCT)
1               0.2125
```



```
   AVG(NVL(COMMISSION_PCT,0))
1                      0.0425
```

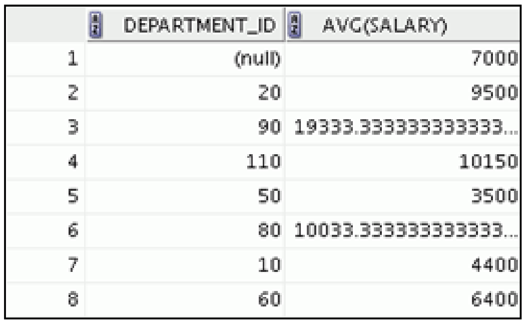
In the first statement, AVG function ignore null values in the column commission_pct while calculating the average. On the other hand, in the second statement, the NVL function forces AVG function to include null values too.

Activity 5: Creating Groups of Data

When using the GROUP BY clause, we need to make sure that all columns in the SELECT list that are not group functions are included in the GROUP BY clause. In this activity, we are going to display the department number and the average salary for each department

```
SELECT      department_id, AVG(salary)
FROM        employees
GROUP BY    department_id ;
```

Output



	DEPARTMENT_ID	AVG(SALARY)
1	(null)	7000
2	20	9500
3	90	19333.333333333333...
4	110	10150
5	50	3500
6	80	10033.333333333333...
7	10	4400
8	60	6400

Here is how this SELECT statement, containing a GROUP BY clause, is evaluated:

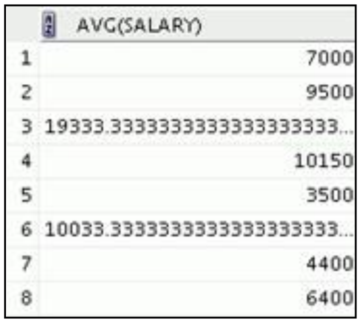
- The SELECT clause specifies the columns to be retrieved, as follows: -
    - Department number column in the EMPLOYEES table
    - The average of all salaries in the group that you specified in the GROUP BY clause
- The FROM clause specifies the tables that the database must access: the EMPLOYEES table.
- The WHERE clause specifies the rows to be retrieved. Because there is no WHERE clause, all rows are retrieved by default.
- The GROUP BY clause specifies how the rows should be grouped. The rows are grouped by department number, so the AVG function that is applied to the salary column calculates the average salary for each department.

Note: To order the query results in ascending or descending order, include the ORDER BY clause in the query.

It is also important to note that the GROUP BY column does not have to be in the SELECT clause. For example, the following SELECT statement displays the average salaries for each department without displaying the respective department numbers. Without the department numbers, however, the results do not look meaningful. You can also use the group function in the ORDER BY clause.

```
SELECT      AVG(salary)
FROM        employees
```

```
GROUP BY    department_id ;
```



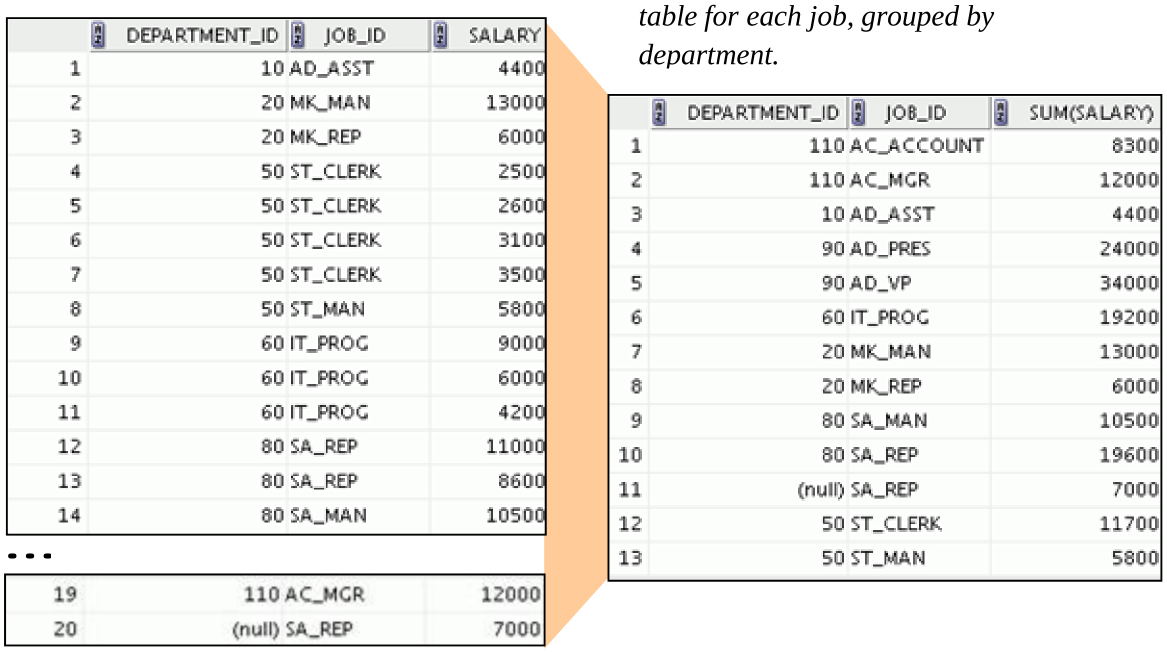
	AVG(SALARY)
1	7000
2	9500
3	19333.3333333333333333333...
4	10150
5	3500
6	10033.3333333333333333333...
7	4400
8	6400

Activity 6: Grouping by more than One Column

Sometimes, we need to see results for groups within groups. Given below is a report (right table) that displays the total salary that is paid to each job title in each department.

EMPLOYEES

Sum the salaries in the EMPLOYEES table for each job, grouped by department.



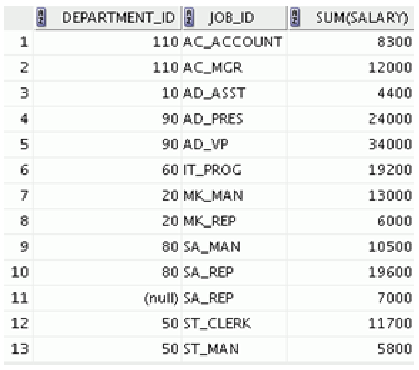
	DEPARTMENT_ID	JOB_ID	SALARY
1	10	AD_ASST	4400
2	20	MK_MAN	13000
3	20	MK_REP	6000
4	50	ST_CLERK	2500
5	50	ST_CLERK	2600
6	50	ST_CLERK	3100
7	50	ST_CLERK	3500
8	50	ST_MAN	5800
9	60	IT_PROG	9000
10	60	IT_PROG	6000
11	60	IT_PROG	4200
12	80	SA_REP	11000
13	80	SA_REP	8600
14	80	SA_MAN	10500
...			
19	110	AC_MGR	12000
20	(null)	SA_REP	7000

	DEPARTMENT_ID	JOB_ID	SUM(SALARY)
1	110	AC_ACCOUNT	8300
2	110	AC_MGR	12000
3	10	AD_ASST	4400
4	90	AD_PRES	24000
5	90	AD_VP	34000
6	60	IT_PROG	19200
7	20	MK_MAN	13000
8	20	MK_REP	6000
9	80	SA_MAN	10500
10	80	SA_REP	19600
11	(null)	SA_REP	7000
12	50	ST_CLERK	11700
13	50	ST_MAN	5800

In the report, the EMPLOYEES table is grouped first by the department number, and then by the job title within that grouping. For example, the four stock clerks in department 50 are grouped together, and a single result (total salary) is produced for all stock clerks in the group. Here's the SELECT statement that returns the result shown in the report:

```
SELECT      department_id, job_id, sum(salary)
FROM        employees
GROUP BY    department_id, job_id
ORDER BY    job_id;
```

Output



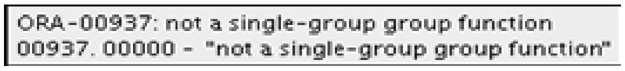
	DEPARTMENT_ID	JOB_ID	SUM(SALARY)
1	110	AC_ACCOUNT	8300
2	110	AC_MGR	12000
3	10	AD_ASST	4400
4	90	AD_PRES	24000
5	90	AD_VP	34000
6	60	IT_PROG	19200
7	20	MK_MAN	13000
8	20	MK_REP	6000
9	80	SA_MAN	10500
10	80	SA_REP	19600
11	(null)	SA_REP	7000
12	50	ST_CLERK	11700
13	50	ST_MAN	5800

Activity 7: Illegal Queries Using Group Functions

In the activity, we are going to demonstrate the errors commonly made while writing queries using group functions. Whenever you use a mixture of individual items (DEPARTMENT_ID) and group functions (COUNT) in the same SELECT statement, you must include a GROUP BY clause that specifies the individual items (in this case, DEPARTMENT_ID). If the GROUP BY clause is missing, the error message "not a single-group group function" appears and an asterisk (*) points to the offending column.

```
SELECT     department_id, COUNT(last_name)
FROM       employees;
```

Output



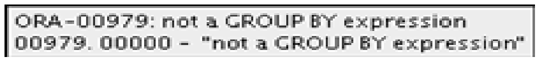
```
ORA-00937: not a single-group group function
00937. 00000 -  "not a single-group group function"
```

The above error can be corrected by adding the GROUP BY clause as:

```
SELECT     department_id, COUNT(last_name)
FROM       employees
GROUP BY   department_id;
```

Any column or expression in the SELECT list that is not an aggregate function must be in the GROUP BY clause. In the following example, job_id is neither in the GROUP BY clause nor is it being used by a group function, so there is a "not a GROUP BY expression" error.

```
SELECT     department_id, job_id, COUNT(last_name)
FROM       employees
GROUP BY   department_id;
```



```
ORA-00979: not a GROUP BY expression
00979. 00000 -  "not a GROUP BY expression"
```

The above error can be corrected by adding job_id in the GROUP BY clause as:

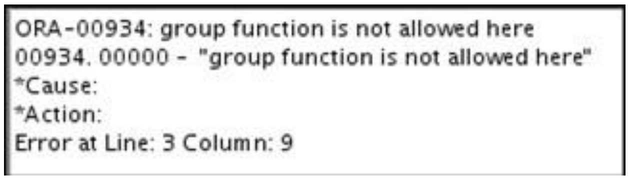
```
SELECT     department_id, job_id, COUNT(last_name)
FROM       employees
GROUP BY   department_id, job_id;
```

Activity 8: Restricting Group Results using HAVING Clause

The WHERE clause cannot be used to restrict groups. The SELECT statement in the following example results in an error because it uses the WHERE clause to restrict the display of the average salaries of those departments that have an average salary greater than $8,000.

```
SELECT     department_id, AVG(salary)
FROM       employees
WHERE      AVG(salary) > 8000
GROUP BY   department_id;
```

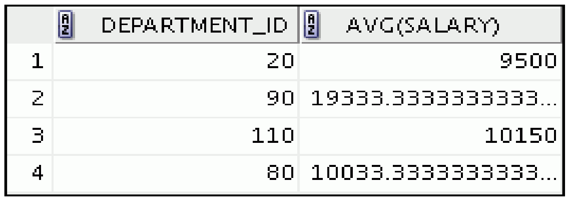
Output



```
ORA-00934: group function is not allowed here
00934. 00000 -  "group function is not allowed here"
*Cause:
*Action:
Error at Line: 3 Column: 9
```

However, it can be corrected the by using the HAVING clause to restrict groups:

```
SELECT     department_id, AVG(salary)
FROM       employees
GROUP BY   department_id
HAVING     AVG(salary) > 8000
```

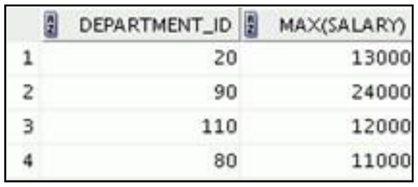


	DEPARTMENT_ID	AVG(SALARY)
1	20	9500
2	90	19333.3333333333...
3	110	10150
4	80	10033.3333333333...

The above statement uses the HAVING clause to restrict groups in the same way that we used the WHERE clause to restrict the rows. It finds the average salary in each of the departments that have an average salary greater than $8,000.
Here' another statement using HAVING clause to display the department numbers and maximum salaries for those departments with a maximum salary greater than $10,000.

```
SELECT     department_id, MAX(salary)
FROM       employees
GROUP BY   department_id
```

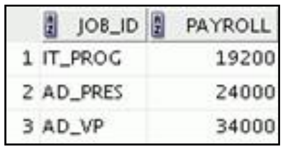
```
HAVING      MAX(salary) > 10000
```



	DEPARTMENT_ID	MAX(SALARY)
1	20	13000
2	90	24000
3	110	12000
4	80	11000

Here’s one more statement that uses both WHERE and HAVING clauses to display the job ID and total monthly salary for each job that has a total payroll exceeding $13,000. It excludes sales representatives and sorts the list by the total monthly salary.

```
SELECT    job_id, SUM(salary) PAYROLL
FROM      employees
WHERE     job_id NOT LIKE '%REP%'
GROUP BY  job_id
HAVING    SUM(salary) > 13000
ORDER BY  SUM(salary);
```



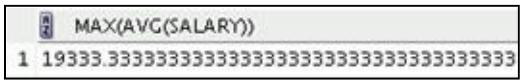
	JOB_ID	PAYROLL
1	IT_PROG	19200
2	AD_PRES	24000
3	AD_VP	34000

Activity 9: Nesting Group Functions

Group functions can be nested to a depth of two functions. In this activity, we are going to calculate the average salary for each department_id and then display the maximum average salary. Note that the GROUP BY clause is mandatory when nesting group functions.

```
SELECT      MAX(AVG(salary))
FROM        employees
GROUP BY    department_id ;
```

Output



	MAX(AVG(SALARY))
1	19333.3333333333333333333333333333333333

3)Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Lab Task 1

Write a SQL statement that displays the highest, lowest, sum, and average salary of all employees. Label the columns as Maximum, Minimum, Sum, and Average, respectively. Round your results to the nearest whole number.

Lab Task 2

Write a SQL statement to display the number of people with the same job.

Lab Task 3

Write a SQL statement to determine the number of managers without listing them. Label the column as Number of Managers.

Lab Task 4

Write a SQL statement that finds the difference between the highest and lowest salaries. Label the column DIFFERENCE.

Lab Task 5

Write a SQL statement that displays the manager number and the salary of the lowest-paid employee for that manager. Exclude anyone whose manager is not known. Exclude any groups where the minimum salary is $6,000 or less. Sort the output in descending order of salary.

Lab Task 6

Write a SQL statement that displays the job, the salary for that job based on department number, and the total salary for that job, for departments 20, 50, 80, and 90, giving each column an appropriate heading.

Lab 06

Displaying Data from Multiple Tables Using Joins

Objective:

This lab explains how to obtain data from more than one table. A join is used to view information from multiple tables. Therefore, you can join tables together to view information from more than one table. More specifically the following various types of joins will be discussed in this lab:

- Natural join
- Join with the USING Clause
- Join with the ON Clause
- Self-join
- Non-equijoins
- OUTER join:
    - LEFT OUTER join
    - RIGHT OUTER join
    - FULL OUTER join
- Cartesian product
    - Cross join

Activity Outcomes:

After completing this lesson, students should be able to do the following:

- Write SELECT statements to access data from more than one table using equijoins and non equijoins
- Join a table to itself by using a self-join
- View data that generally does not meet a join condition by using OUTER joins
- Generate a cartesian product of all rows from two or more tables

Instructor Note:

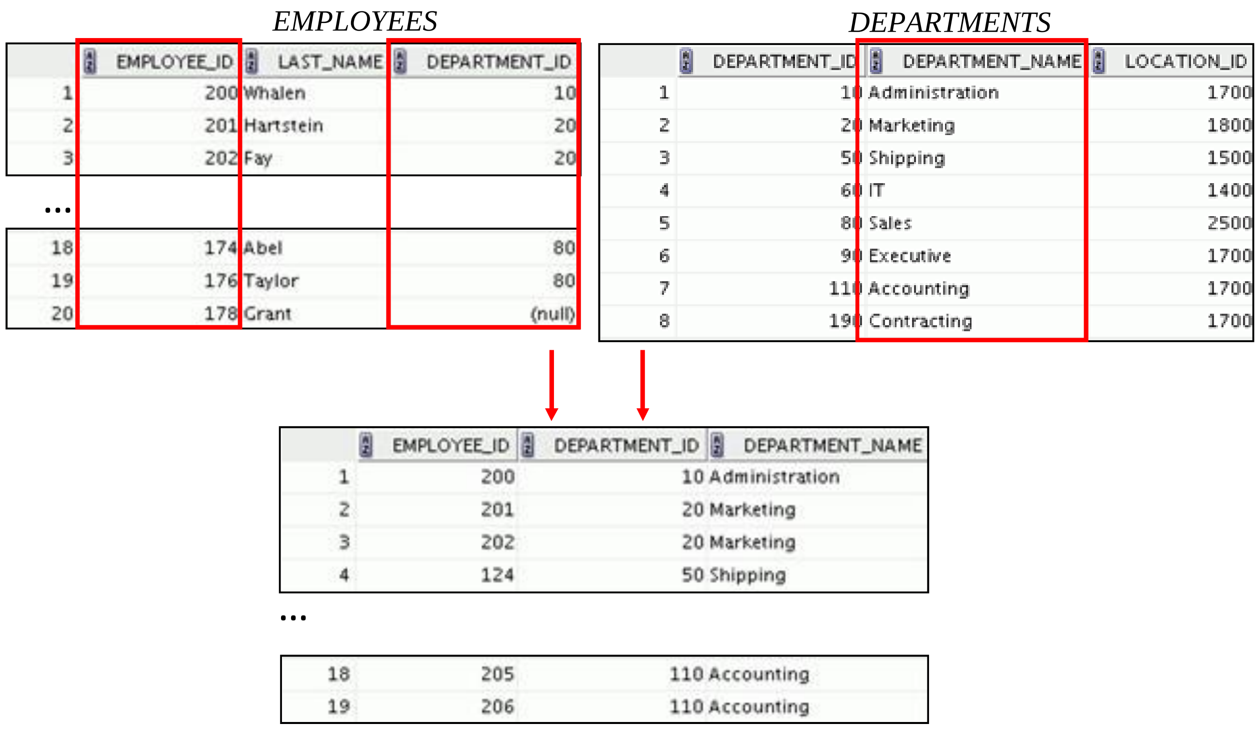
Please make sure that you have completed the previous labs.

1) Useful Concepts

Very often we need to use data from more than one table. In the following example, the report displays data from two separate tables:

- Employee IDs exist in the EMPLOYEES table.
- Department IDs exist in both the EMPLOYEES and DEPARTMENTS tables.
- Department names exist in the DEPARTMENTS table.

To produce the report, we need to join the EMPLOYEES and DEPARTMENTS tables, and access data from both of them.


EMPLOYEES
DEPARTMENTS

	EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
1	200	Whalen	10
2	201	Hartstein	20
3	202	Fay	20
…			
18	174	Abel	80
19	176	Taylor	80
20	178	Grant	(null)

	DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
1	10	Administration	1700
2	20	Marketing	1800
3	50	Shipping	1500
4	60	IT	1400
5	80	Sales	2500
6	90	Executive	1700
7	110	Accounting	1700
8	190	Contracting	1700

	EMPLOYEE_ID	DEPARTMENT_ID	DEPARTMENT_NAME
1	200	10	Administration
2	201	20	Marketing
3	202	20	Marketing
4	124	50	Shipping
…			
18	205	110	Accounting
19	206	110	Accounting

To join tables, the following join syntax is used that is compliant with the SQL:1999 standard:

```
SELECT      table1.column, table2.column
FROM        table1
            [NATURAL JOIN table2] |
            [JOIN table2 USING (column_name)] |
            [JOIN table2 ON (table1.column_name = table2.column_name)]|
            [LEFT|RIGHT|FULL OUTER JOIN table2
            ON (table1.column_name = table2.column_name)]|
            [CROSS JOIN table2];
```

In the above syntax:

- table1.column denotes the table and the column from which data is retrieved
- NATURAL JOIN joins two tables based on the same column name
- JOIN table2 USING column_name performs an equijoin based on the column name

- JOIN table2 ON table1.column_name = table2.column_name performs an equijoin based on the condition in the ON clause
- LEFT/RIGHT/FULL OUTER is used to perform OUTER joins
- CROSS JOIN returns a Cartesian product from the two tables

When joining two or more tables, you need to qualify the names of the columns with the table name to avoid ambiguity. Without the table prefixes, the DEPARTMENT_ID column in the SELECT list could be from either the DEPARTMENTS table or the EMPLOYEES table. It is necessary to add the table prefix to execute your query. If there are no common column names between the two tables, there is no need to qualify the columns. However, using the table prefix improves performance, because you tell the Oracle server exactly where to find the columns.

However, qualifying column names with table names can be time consuming, particularly if the table names are lengthy. Instead, you can use table aliases. Just as a column alias gives a column another name, a table alias gives a table another name. Table aliases help to keep SQL code smaller, therefore, using less memory.

The table name is specified in full, followed by a space, and then the table alias. For example, the EMPLOYEES table can be given an alias of *e*, and the DEPARTMENTS table an alias of *d*.

Guidelines:

- Table aliases can be up to 30 characters in length, but shorter aliases are better than longer ones.
- If a table alias is used for a particular table name in the FROM clause, that table alias must be substituted for the table name throughout the SELECT statement.
- Table aliases should be meaningful.
- The table alias is valid for only the current SELECT statement.

Natural Join

The NATURAL JOIN clause is based on all the columns in the two tables that have the same name. It selects rows from the two tables that have equal values in all matched columns. If the columns having the same names have different data types, an error is returned.

Join with the USING Clause

If several columns have the same names but the data types do not match, we can use the USING clause to specify the columns for the equijoin i.e., values in the common column in both the tables must be equal. We can also use the USING clause to match only specific column(s) when more than one column matches.

Note: The NATURAL JOIN and USING clauses are mutually exclusive i.e., both cannot be used in a single statement.

Join with the ON Clause

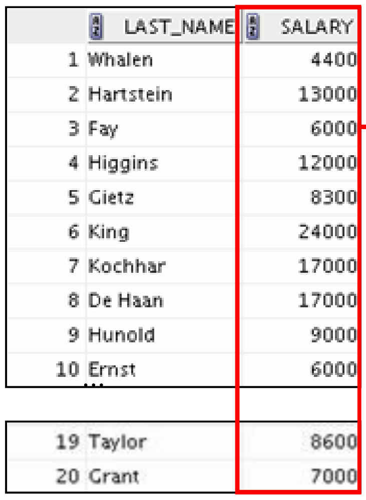
As opposed to the natural join where the join condition is basically an equijoin of all columns with the same name, the ON clause can specify arbitrary conditions or specify columns to join. With ON clause, the join condition is separated from any other search or filter conditions in the WHERE clause and, consequently, makes the code easy to understand.

Non-equijoins

A non-equijoin is a join condition containing something other than an equality operator. The relationship between the EMPLOYEES table and the JOB_GRADES table is an example of a non-equijoin. The

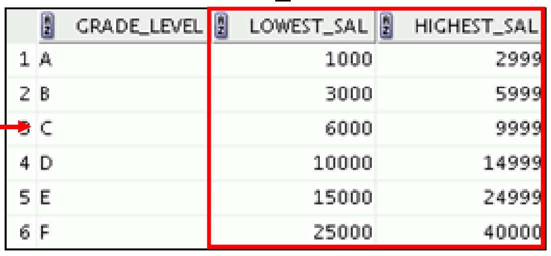
SALARY column in the EMPLOYEES table ranges between the values in the LOWEST_SAL and HIGHEST_SAL columns of the JOB_GRADES table. Therefore, each employee can be graded based on their salary. The relationship is obtained using an operator other than the equality (=) operator.

EMPLOYEES



	LAST_NAME	SALARY
1	Whalen	4400
2	Hartstein	13000
3	Fay	6000
4	Higgins	12000
5	Gietz	8300
6	King	24000
7	Kochhar	17000
8	De Haan	17000
9	Hunold	9000
10	Ernst	6000
…		
19	Taylor	8600
20	Grant	7000

JOB_GRADES



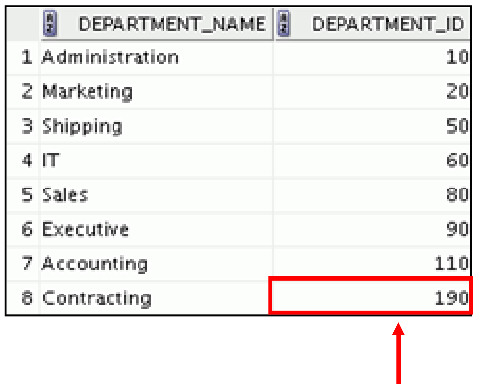
	GRADE_LEVEL	LOWEST_SAL	HIGHEST_SAL
1	A	1000	2999
2	B	3000	5999
3	C	6000	9999
4	D	10000	14999
5	E	15000	24999
6	F	25000	40000

The JOB_GRADES table defines the LOWEST_SAL and HIGHEST_SAL range of values for each GRADE_LEVEL. Therefore, the GRADE_LEVEL column can be used to assign grades to each employee

Outer Joins

If a row does not satisfy a join condition, the row does not appear in the query result. In the following example, a simple equijoin condition is used on the EMPLOYEES and DEPARTMENTS tables to return the result on the right.

DEPARTMENTS

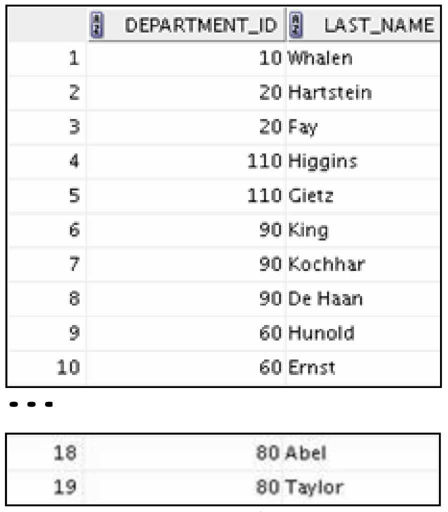


	DEPARTMENT_NAME	DEPARTMENT_ID
1	Administration	10
2	Marketing	20
3	Shipping	50
4	IT	60
5	Sales	80
6	Executive	90
7	Accounting	110
8	Contracting	190

There are no employees in department 190.

Employee "Grant" has not been assigned a department ID.

Equijoin with EMPLOYEES



	DEPARTMENT_ID	LAST_NAME
1	10	Whalen
2	20	Hartstein
3	20	Fay
4	110	Higgins
5	110	Gietz
6	90	King
7	90	Kochhar
8	90	De Haan
9	60	Hunold
10	60	Ernst
…		
18	80	Abel
19	80	Taylor

The result set does not contain the following:

- Department ID 190, because there are no employees with that department ID recorded in the EMPLOYEES table

- The employee with the last name of Grant, because this employee has not been assigned a department ID

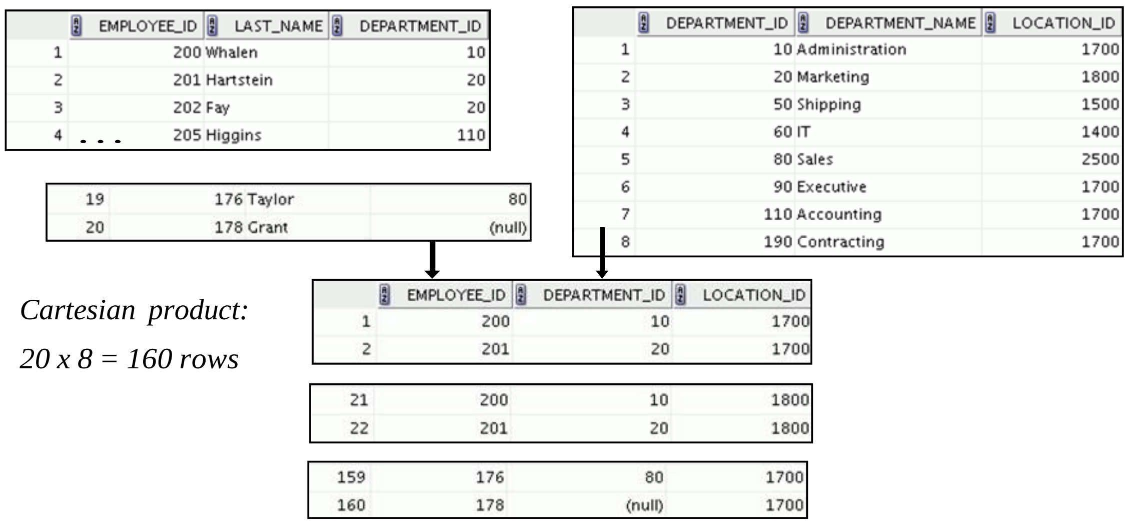
To return the department record that does not have any employees, or employees that do not have an assigned department, you can use an OUTER join.
In SQL:1999, the join of two tables returning only matched rows is called an INNER join. There are three types of OUTER joins:

- LEFT OUTER
- RIGHT OUTER
- FULL OUTER

A join between two tables that returns the results of the INNER join as well as the unmatched rows from the left (or right) table is called a left (or right) OUTER join. A join between two tables that returns the results of an INNER join as well as the results of a left and right join is a full OUTER join.

Cartesian product
When a join condition is invalid or omitted completely, the result is a Cartesian product, in which all combinations of rows are displayed. All rows in the first table are joined to all rows in the second table. A Cartesian product tends to generate a large number of rows and the result is rarely useful. You should, therefore, always include a valid join condition unless you have a specific need to combine all rows from all tables. Cartesian products are useful for some tests when you need to generate a large number of rows to simulate a reasonable amount of data.

EMPLOYEES (20 rows) *DEPARTMENTS(8 rows)*



	EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
1	200	Whalen	10
2	201	Hartstein	20
3	202	Fay	20
4	205	Higgins	110
. . .			
19	176	Taylor	80
20	178	Grant	(null)

	DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
1	10	Administration	1700
2	20	Marketing	1800
3	50	Shipping	1500
4	60	IT	1400
5	80	Sales	2500
6	90	Executive	1700
7	110	Accounting	1700
8	190	Contracting	1700

Cartesian product:
20 x 8 = 160 rows

	EMPLOYEE_ID	DEPARTMENT_ID	LOCATION_ID
1	200	10	1700
2	201	20	1700
21	200	10	1800
22	201	20	1800
159	176	80	1700
160	178	(null)	1700

The above example displays the employee's last name and the department name from the EMPLOYEES and DEPARTMENTS tables. Because no join condition was specified, all rows (20 rows) from the EMPLOYEES table are joined with all rows (8 rows) in the DEPARTMENTS table, thereby generating 160 rows in the output.

2) Solved Lab Activites

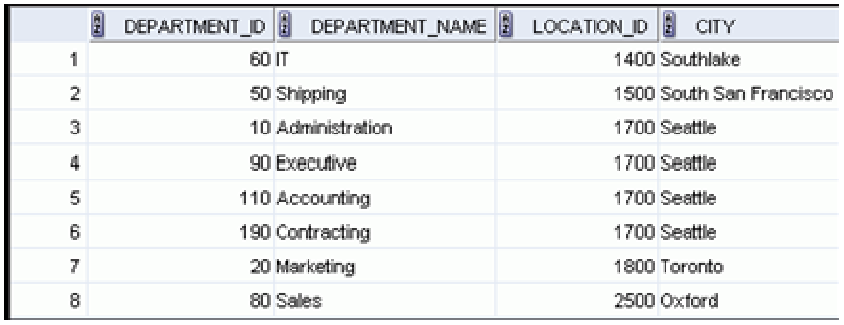
Activity No	*Allocated Time*	*Level of complexity*	*CLO Mapping*
Activity 1	***5 mins***	***Low***	***CLO-5***
Activity 2	***5 mins***	***Low***	***CLO-5***
Activity 3	***5 mins***	***Low***	***CLO-5***
Activity 4	***5 mins***	***Low***	***CLO-5***
Activity 5	***10 mins***	***Medium***	***CLO-5***
Activity 6	***10 mins***	***Medium***	***CLO-5***
Activity 7	***10 mins***	***Medium***	***CLO-5***
Activity 8	***10 mins***	***Medium***	***CLO-5***
Activity 9	***5 mins***	***Low***	***CLO-5***
Activity 10	***5 mins***	***Low***	***CLO-5***
Activity 11	***5 mins***	***Low***	***CLO-5***
Activity 12	***5 mins***	***Low***	***CLO-5***

Activity 1: Creating Natural Joins

In this activity, the LOCATIONS table is joined to the DEPARTMENT table by the LOCATION_ID column, which is the only column of the same name in both tables. If other common columns were present, the join would have used them all.

```
SELECT   department_id, department_name, location_id, city
FROM     departments NATURAL JOIN locations ;
```

Output



```
   DEPARTMENT_ID DEPARTMENT_NAME   LOCATION_ID CITY
1             60 IT                       1400 Southlake
2             50 Shipping                 1500 South San Francisco
3             10 Administration           1700 Seattle
4             90 Executive                1700 Seattle
5            110 Accounting               1700 Seattle
6            190 Contracting              1700 Seattle
7             20 Marketing                1800 Toronto
8             80 Sales                    2500 Oxford
```

Additional restrictions on a natural join are implemented by using a WHERE clause. The following example limits the rows of output to those with a department ID equal to 20 or 50:

```
SELECT   department_id, department_name, location_id, city
FROM     departments NATURAL JOIN locations
WHERE    department_id IN (20, 50);
```

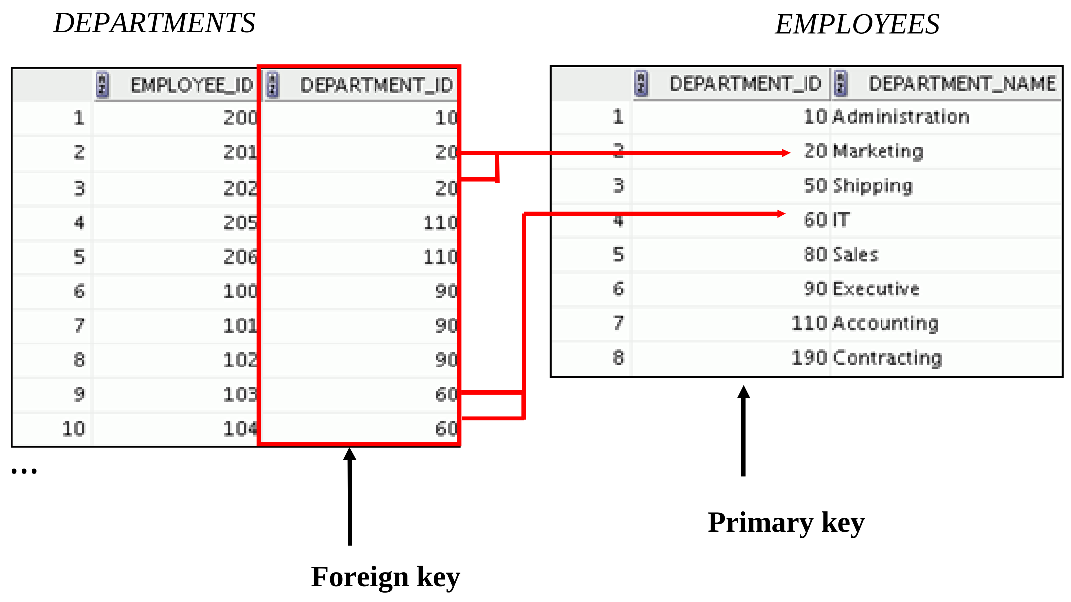
Activity 2: Creating Joins with the USING Clause

In the activity we are going to write a query to determine an employee's department name. To do that, we need to compare the value in the DEPARTMENT_ID column in the EMPLOYEES table with the DEPARTMENT_ID values in the DEPARTMENTS table. The relationship between the EMPLOYEES and DEPARTMENTS tables is an equijoin; that is, values in the DEPARTMENT_ID column in both the tables must be equal. Frequently, this type of join involves primary and foreign key complements.

Note: Equijoins are also called simple joins or inner joins

```
SELECT   employee_id, department_name
FROM     employees JOIN    departments USING (department_id);
```

Output

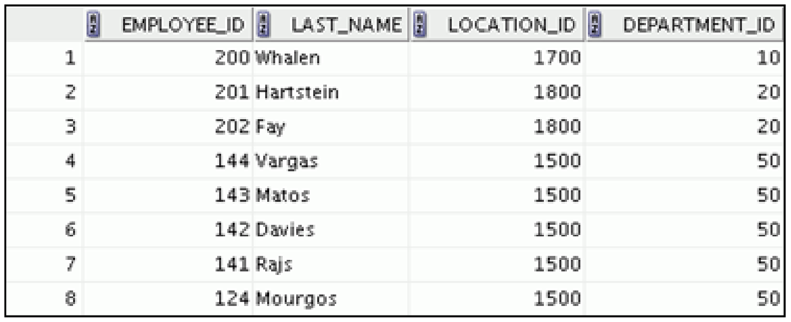


	EMPLOYEE_ID	DEPARTMENT_ID
1	200	10
2	201	20
3	202	20
4	205	110
5	206	110
6	100	90
7	101	90
8	102	90
9	103	60
10	104	60

	DEPARTMENT_ID	DEPARTMENT_NAME
1	10	Administration
2	20	Marketing
3	50	Shipping
4	60	IT
5	80	Sales
6	90	Executive
7	110	Accounting
8	190	Contracting

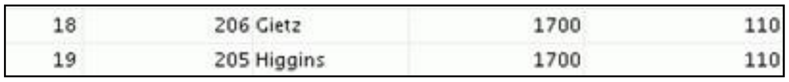
In the following query, the DEPARTMENT_ID columns in the EMPLOYEES and DEPARTMENTS tables are joined and thus the LOCATION_ID of the department where an employee works is shown.

```
SELECT   employee_id, last_name, location_id, department_id
FROM     employees JOIN departments USING (department_id);
```



	EMPLOYEE_ID	LAST_NAME	LOCATION_ID	DEPARTMENT_ID
1	200	Whalen	1700	10
2	201	Hartstein	1800	20
3	202	Fay	1800	20
4	144	Vargas	1500	50
5	143	Matos	1500	50
6	142	Davies	1500	50
7	141	Rajs	1500	50
8	124	Mourgos	1500	50

…



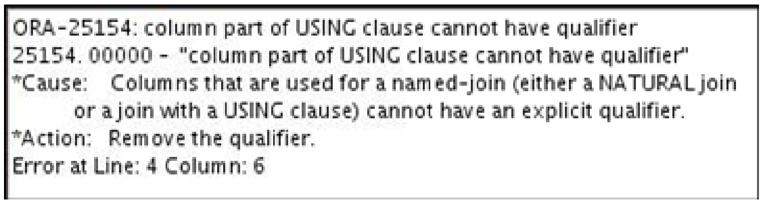
18	206 Gietz	1700	110
19	205 Higgins	1700	110

Activity 3: Using Table Aliases with the USING Clause

When joining with the USING clause, we cannot qualify a column that is used in the USING clause itself. Furthermore, if that column is used anywhere in the SQL statement, you cannot alias it. For example, in the following query, you should not alias the location_id column in the WHERE clause because the column is used in the USING clause. The columns that are referenced in the USING clause should not have a qualifier (table name or alias) anywhere in the SQL statement.

```
SELECT     l.city, d.department_name
FROM       locations l JOIN departments d USING(location_id)
WHERE      d.location_id = 1400;
```

Output



```
ORA-25154: column part of USING clause cannot have qualifier
25154. 00000 - "column part of USING clause cannot have qualifier"
*Cause:    Columns that are used for a named-join (either a NATURAL join
           or a join with a USING clause) cannot have an explicit qualifier.
*Action:   Remove the qualifier.
Error at Line: 4 Column: 6
```

The following statement is valid:

```
SELECT     l.city, d.department_name
FROM       locations l JOIN departments d USING(location_id)
WHERE      location_id = 1400;
```

The columns that are common in both the tables, but not used in the USING clause, must be prefixed with a table alias (e.g., city, department_name in the above statement); otherwise, you get the "column ambiguously defined" error.

Activity 4: Creating Joins with the ON Clause

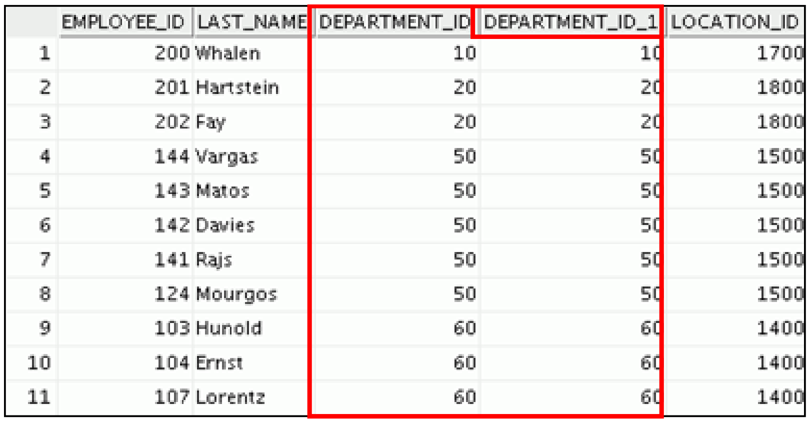
In this activity, the DEPARTMENT_ID columns in the EMPLOYEES and DEPARTMENTS table are joined using the ON clause. Wherever a department ID in the EMPLOYEES table equals a department ID in the DEPARTMENTS table, the row is returned. The table alias is necessary to qualify the matching column_names. You can also use the ON clause to join columns that have

different names. The parenthesis around the joined columns (e.department_id = d.department_id) is optional. So, even ON e.department_id = d.department_id will work.

Note: When you use the Execute Statement icon to run the query, SQL Developer suffixes a '_1' to differentiate between the two department_ids.

```
SELECT   e.employee_id, e.last_name, e.department_id, d.department_id,
         d.location_id
FROM     employees e JOIN departments d
ON       (e.department_id = d.department_id);
```

Output



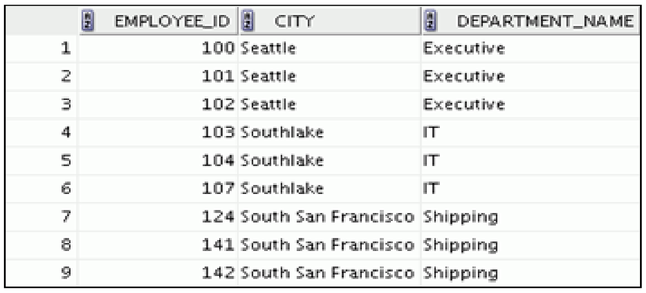
	EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_ID_1	LOCATION_ID
1	200	Whalen	10	10	1700
2	201	Hartstein	20	20	1800
3	202	Fay	20	20	1800
4	144	Vargas	50	50	1500
5	143	Matos	50	50	1500
6	142	Davies	50	50	1500
7	141	Rajs	50	50	1500
8	124	Mourgos	50	50	1500
9	103	Hunold	60	60	1400
10	104	Ernst	60	60	1400
11	107	Lorentz	60	60	1400

Activity 5: Creating Three-Way Joins with the ON Clause

When using the GROUP BY clause, we need to make sure that all columns in the SELECT list that are not group functions are included in the GROUP BY clause. In this activity, we are going to display the department number and the average salary for each department

```
SELECT     employee_id, city, department_name FROM employees e
JOIN       departments d
ON         d.department_id = e.department_id JOIN  locations l
ON         d.location_id = l.location_id;
```

Output



	EMPLOYEE_ID	CITY	DEPARTMENT_NAME
1	100	Seattle	Executive
2	101	Seattle	Executive
3	102	Seattle	Executive
4	103	Southlake	IT
5	104	Southlake	IT
6	107	Southlake	IT
7	124	South San Francisco	Shipping
8	141	South San Francisco	Shipping
9	142	South San Francisco	Shipping

…

The same three-way join can also be accomplished with the USING clause

```
SELECT      e.employee_id, l.city, d.department_name
FROM        employees e
JOIN        departments d
USING       (department_id)
JOIN        locations l
USING       (location_id);
```

Activity 6: Applying Additional Conditions to a Join

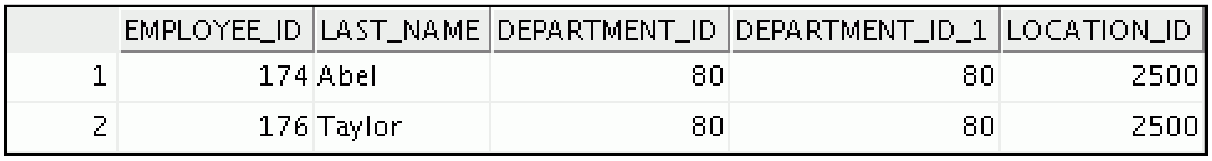
We can apply additional conditions to the join. In the lab activity, we are going to perform a join on the EMPLOYEES and DEPARTMENTS tables and, in addition, display only employees who have a manager ID of 149. To add additional conditions to the ON clause, we can add AND clauses. Alternatively, you can use a WHERE clause to apply additional conditions. Both the queries produce the same output.

```
 SELECT     e.employee_id, e.last_name, e.department_id,
            d.department_id, d.location_id
FROM        employees e JOIN departments d
ON          e.department_id = d.department_id
AND         e.manager_id = 149 ;
```

OR

```
SELECT      e.employee_id, e.last_name, e.department_id,
            d.department_id, d.location_id
FROM        employees e JOIN departments d
ON          e.department_id = d.department_id
WHERE       e.manager_id = 149 ;
```

Output



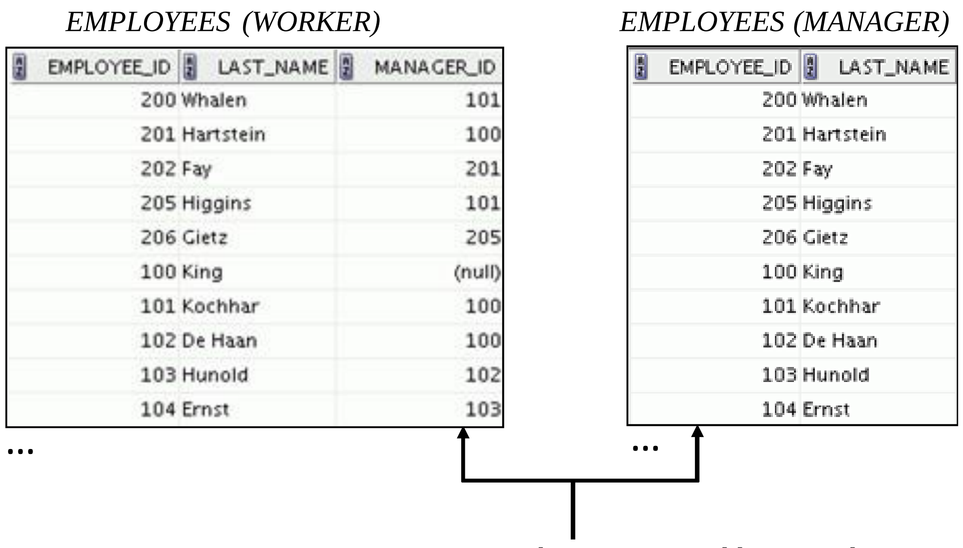
	EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_ID_1	LOCATION_ID
1	174	Abel	80	80	2500
2	176	Taylor	80	80	2500

Activity 7: Self-Join: Joining a Table to Itself

Sometimes we need to join a table to itself. For example, to find the name of each employee's manager, we need to join the EMPLOYEES table to itself, or perform a self-join.
To find the name of Ernst's manager, we need to:

- ***Find Ernst in the EMPLOYEES table by looking at the LAST_NAME column***
- ***Find the manager number for Ernst by looking at the MANAGER_ID column. Ernst's manager number is 103.***
- ***Find the name of the manager with EMPLOYEE_ID 103 by looking at the LAST_NAME column. Hunold's employee number is 103, so Hunold is Ernst's manager.***

In this process, we look in the table twice. The first time we look in the table to find Ernst in the LAST_NAME column and the MANAGER_ID value of 103. The second time we look in the EMPLOYEE_ID column to find 103 and the LAST_NAME column to find Hunold.



EMPLOYEES (WORKER)

EMPLOYEE_ID	LAST_NAME	MANAGER_ID
200	Whalen	101
201	Hartstein	100
202	Fay	201
205	Higgins	101
206	Gietz	205
100	King	(null)
101	Kochhar	100
102	De Haan	100
103	Hunold	102
104	Ernst	103

EMPLOYEES (MANAGER)

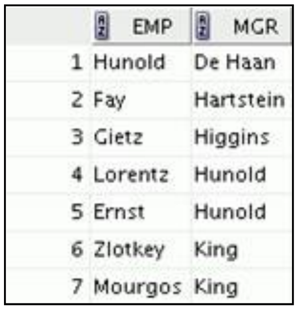
EMPLOYEE_ID	LAST_NAME
200	Whalen
201	Hartstein
202	Fay
205	Higgins
206	Gietz
100	King
101	Kochhar
102	De Haan
103	Hunold
104	Ernst

MANAGER_ID in the WORKER table is equal to EMPLOYEE_ID in the MANAGER table.

In this lab activity, we are going to perform self-join of the EMPLOYEE table for the above scenario. The following statement shows a self-join of the EMPLOYEES table, based on the EMPLOYEE_ID and MANAGER_ID columns.

```
SELECT     worker.last_name emp, manager.last_name mgr
FROM       employees worker JOIN employees manager
ON         (worker.manager_id = manager.employee_id);
```

Output



	EMP	MGR
1	Hunold	De Haan
2	Fay	Hartstein
3	Gietz	Higgins
4	Lorentz	Hunold
5	Ernst	Hunold
6	Zlotkey	King
7	Mourgos	King

Activity 8: Retrieving Records with Non-equijoins

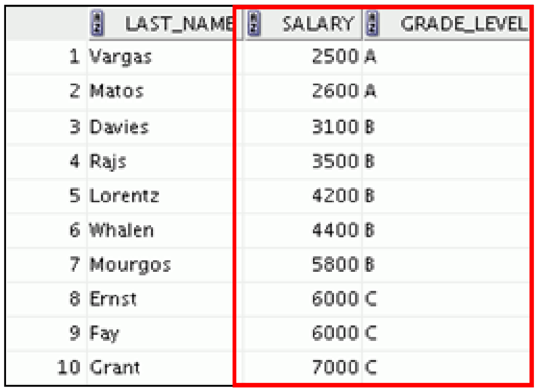
In this lab activity, we are going to create a non-equijoin to evaluate an employee's salary grade. The salary must be between any pair of the low and high salary ranges. It is important to note that all employees appear exactly once when this query is executed. No employee is repeated in the list. There are two reasons for this:

- ***None of the rows in the JOB_GRADES table contain grades that overlap. That is, the salary value for an employee can lie only between the low salary and high salary values of one of the rows in the salary grade table.***
- ***All of the employees' salaries lie within the limits provided by the job grade table. That is, no employee earns less than the lowest value contained in the LOWEST_SAL column or more than the highest value contained in the HIGHEST_SAL column.***

Note: Other conditions (such as <= and >=) can be used, but BETWEEN is the simplest. Remember to specify the low value first and the high value last when using the BETWEEN condition. The Oracle server translates the BETWEEN condition to a pair of AND conditions. Therefore, using BETWEEN has no performance benefits, but should be used only for logical simplicity.

```
SELECT     e.last_name, e.salary, j.grade_level
FROM       employees e JOIN job_grades j
ON         e.salary BETWEEN j.lowest_sal AND j.highest_sal;
```

Output



	LAST_NAME	SALARY	GRADE_LEVEL
1	Vargas	2500	A
2	Matos	2600	A
3	Davies	3100	B
4	Rajs	3500	B
5	Lorentz	4200	B
6	Whalen	4400	B
7	Mourgos	5800	B
8	Ernst	6000	C
9	Fay	6000	C
10	Grant	7000	C

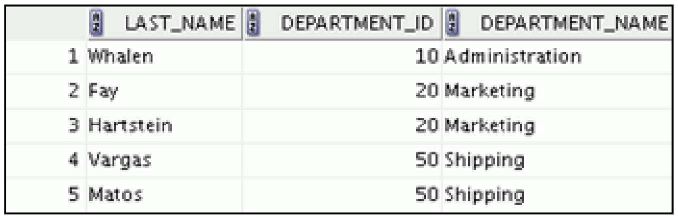
…

Activity 9: Retrieving data using LEFT OUTER JOIN

In this activity, we are going to write a query retrieving all the rows in the EMPLOYEES table, which is the left table, even if there is no match in the DEPARTMENTS table.

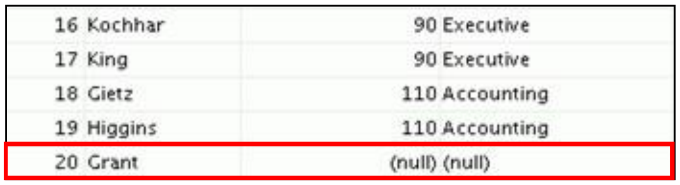
```
SELECT     e.last_name, e.department_id, d.department_name
FROM       employees e   LEFT OUTER JOIN   departments d
ON         (e.department_id = d.department_id) ;
```

Output



	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
1	Whalen	10	Administration
2	Fay	20	Marketing
3	Hartstein	20	Marketing
4	Vargas	50	Shipping
5	Matos	50	Shipping

…



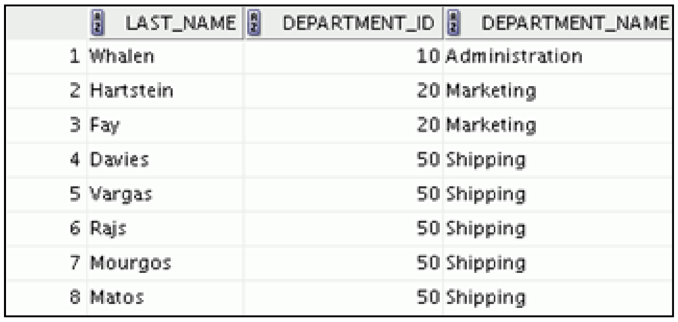
16	Kochhar	90	Executive
17	King	90	Executive
18	Gietz	110	Accounting
19	Higgins	110	Accounting
20	Grant	(null)	(null)

Activity 10: Retrieving data using RIGHT OUTER JOIN

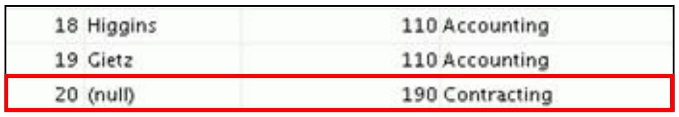
In this activity, we are going to write a query retrieving all the rows in the DEPARTMENTS table, which is the table at the right, even if there is no match in the EMPLOYEES table.

```
SELECT      e.last_name, d.department_id, d.department_name
FROM        employees e RIGHT OUTER JOIN departments d
ON          (e.department_id = d.department_id) ;
```

Output



	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
1	Whalen	10	Administration
2	Hartstein	20	Marketing
3	Fay	20	Marketing
4	Davies	50	Shipping
5	Vargas	50	Shipping
6	Rajs	50	Shipping
7	Mourgos	50	Shipping
8	Matos	50	Shipping



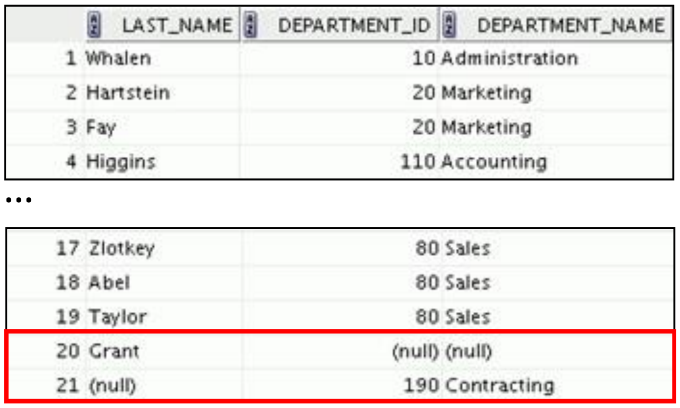
18	Higgins	110	Accounting
19	Gietz	110	Accounting
20	(null)	190	Contracting

Activity 11: Retrieving data using FULL OUTER JOIN

In this activity, we are going to write a query retrieving all rows in the EMPLOYEES table, even if there is no match in the DEPARTMENTS table. It also retrieves all rows in the DEPARTMENTS table, even if there is no match in the EMPLOYEES table

```
SELECT      e.last_name, d.department_id, d.department_name
FROM        employees e FULL OUTER JOIN departments d
ON          (e.department_id = d.department_id) ;
```

Output



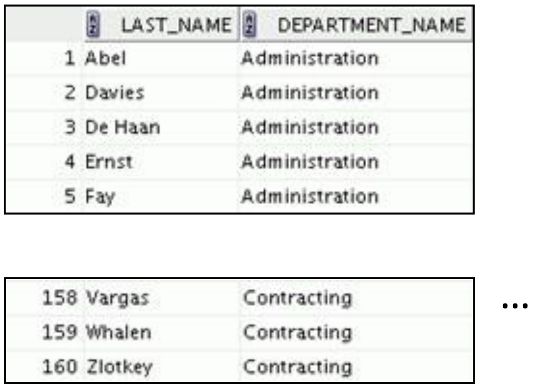
	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
1	Whalen	10	Administration
2	Hartstein	20	Marketing
3	Fay	20	Marketing
4	Higgins	110	Accounting
...			
17	Zlotkey	80	Sales
18	Abel	80	Sales
19	Taylor	80	Sales
20	Grant	(null)	(null)
21	(null)	190	Contracting

Activity 12: Creating Cross Joins

In this activity, we are going to write a query that produces a Cartesian product of the EMPLOYEES and DEPARTMENTS tables.

```
SELECT      last_name, department_name
FROM        employees CROSS JOIN departments ;
```

Output



	LAST_NAME	DEPARTMENT_NAME
1	Abel	Administration
2	Davies	Administration
3	De Haan	Administration
4	Ernst	Administration
5	Fay	Administration
...		
158	Vargas	Contracting
159	Whalen	Contracting
160	Zlotkey	Contracting

It is a good practice to explicitly state CROSS JOIN in your SELECT when you intend to create a Cartesian product. Therefore, it is very clear that you intend for this to happen and it is not the result of missing joins.

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Lab Task 1

Write a SQL statement for HR department to produce the addresses of all the departments. Use the LOCATIONS and COUNTRIES tables. Show the location ID, street address, city, state or province, and country in the output. Use a NATURAL JOIN to produce the results.

Lab Task 2

The HR department needs a report of all employees. Write a SQL statement to display the last name, department number, and department name for all the employees.

Lab Task 3

Write a SQL statement to retrieves all the employees who works in Sales deparment and are from Africa.

Lab Task 4

Write a SQL query to find the employees name, department name, full name (first and last name) of the manager and their city.

Lab Task 5

Write a SQL query to find those employees who joined in 2001. Return job title, department name, employee name, and joining date of the job.

Lab Task 6

The HR department needs a report of employees in Toronto. Write a SQL statement to display the last name, job, department number, and the department name for all employees who work in Toronto.

Lab 07

Using Subqueries to Solve Queries

Objective:

This lab is about more advanced features of the SELECT statement. You can write subqueries in the WHERE, HAVING, FROM clauses of SQL statement to obtain values based on an unknown conditional value. This lab also covers single-row subqueries and multiple-row subqueries.

Activity Outcomes:

After completing this lab, students should be able to do the following:

- Define subqueries
- Describe the types of problems that the subqueries can solve
- List the types of subqueries
- Write single-row and multiple-row subqueries

Instructor Note:

Please make sure that you have completed the previous labs.

1) Useful Concepts

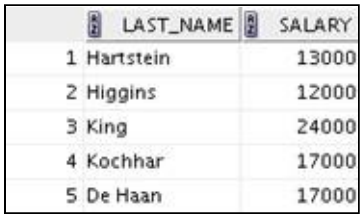
Suppose you want to write a query to find out who earns a salary greater than Abel's salary. To solve this problem, you need two queries: one to find how much Abel earns, and a second query to find who earns more than that amount. You can solve this problem by combining the two queries, placing one query inside the other query. The inner query (or subquery) returns a value that is used by the outer query (or main query). Using a subquery is equivalent to performing two sequential queries and using the result of the first query as the search value in the second query.
A subquery is a SELECT statement that is embedded in the clause of another SELECT statement. You can build powerful statements out of simple ones by using subqueries. They can be very useful when you need to select rows from a table with a condition that depends on the data in the table itself. You can place the subquery in a number of SQL clauses, including the following:

- WHERE clause
- HAVING clause
- FROM clause

```
SELECT     select_list
FROM       table
WHERE      expr operator
                      (SELECT    select_list
                      FROM       table);
```

In the above syntax, *operator* includes a comparison condition which fall into two classes: single-row operators (>, =, >=, <, <>, <=) and multiple-row operators (IN, ANY, ALL, EXISTS). The subquery is often referred to as a nested SELECT, sub-SELECT, or inner SELECT statement. The subquery generally executes first, and its output is used to complete the query condition for the main (or outer) query.
Based on the above syntax, the following statement find out employees who earns a salary greater than Abel's salary:

```
SELECT     last_name, salary
FROM       employees
WHERE      salary >   (SELECT    salary
                      FROM       employees
                      WHERE      last_name = 'Abel');
```



	LAST_NAME	SALARY
1	Hartstein	13000
2	Higgins	12000
3	King	24000
4	Kochhar	17000
5	De Haan	17000

In the above statement, the inner query determines the salary of employee Abel. The outer query takes the result of the inner query and uses this result to display all the employees who earn more than employee Abel.

Rules for using subquery:

- A subquery must be enclosed in parentheses.
- Subquery should be placed the on the right side of the comparison condition for readability. However, the subquery can appear on either side of the comparison operator.
- Two classes of comparison conditions are used in subqueries: single-row operators and multiple-row operators:
    - Single-row subqueries: Queries that return only one row from the inner SELECT statement. They use the following comparison operators:

Operator	Meaning
=	Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
<>	Not equal to

- Multiple-row subqueries: Queries that return more than one row from the inner SELECT statement. They use the following comparison operators:

Operator	Meaning
IN	Equal to any member in the list
ANY	Must be preceded by =, !=, >, <, <=, >=. Returns TRUE if at least one element exists in the result-set of the Subquery for which the relation is TRUE.
ALL	Must be preceded by =, !=, >, <, <=, >=. Returns TRUE if the relation is TRUE for all elements in the result set of the Subquery.
EXISTS	Evaluates to TRUE if the subquery returns at least one row.

2) Solved Lab Activites

Activity No	*Allocated Time*	*Level of complexity*	*CLO Mapping*
Activity 1	***5 mins***	***Low***	***CLO-5***
Activity 2	***5 mins***	***Low***	***CLO-5***
Activity 3	***5 mins***	***Low***	***CLO-5***
Activity 4	***10 mins***	***Medium***	***CLO-5***

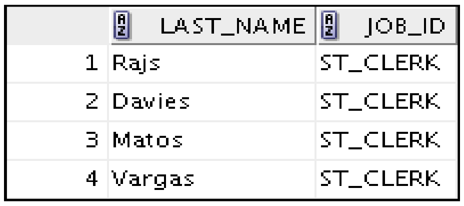
Activity 5	***5 mins***	***Low***	***CLO-5***
Activity 6	***5 mins***	***Low***	***CLO-5***
Activity 7	***5 mins***	***Low***	***CLO-5***
Activity 8	***5 mins***	***Low***	***CLO-5***
Activity 9	***5 mins***	***Low***	***CLO-5***
Activity 10	***5 mins***	***Low***	***CLO-5***

Activity 1: Using single row subquery

Use single row subquery to display the employees whose job ID is the same as that of employee 141

```
SELECT    last_name, job_id
FROM      employees
WHERE     job_id =
                  (SELECT     job_id
                   FROM       employees
                   WHERE      employee_id = 141);
```

Output



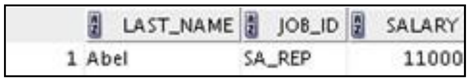
```
     LAST_NAME  JOB_ID
   1 Rajs       ST_CLERK
   2 Davies     ST_CLERK
   3 Matos      ST_CLERK
   4 Vargas     ST_CLERK
```

Activity 2: Using Multiple Single Row Subqueries

In this activity, we are going to display employees who do the same job as "Taylor," but earn more salary than him. This will require three query blocks: the outer query and two inner queries.

```
SELECT    last_name, job_id, salary
FROM      employees
WHERE     job_id =
                  (SELECT     job_id
                   FROM       employees
                   WHERE      last_name = 'Taylor');
AND       salary >
                  (SELECT     salary
                   FROM       employees
                   WHERE      last_name = 'Taylor');
```

Output



```
     LAST_NAME  JOB_ID  SALARY
   1 Abel       SA_REP   11000
```

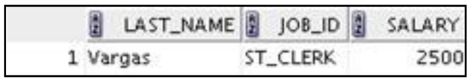
The inner query blocks are executed first, producing the query results corresponding to job_id and salary of Taylor, respectively. The outer query block is then processed and uses the values that were returned by the inner queries to complete its search conditions. Both inner queries return single values, so this SQL statement is called a single-row subquery.

Activity 3: Using Group Functions in a Subquery

In this activity, we are going to display data from a main query by using a group function in a subquery to return a single row. The subquery must be in parentheses and is placed after the comparison condition.

```
SELECT    last_name, job_id, salary
FROM      employees
WHERE     salary =
                  (SELECT     MIN(salary)
                   FROM       employees);
```

Output



	LAST_NAME	JOB_ID	SALARY
1	Vargas	ST_CLERK	2500

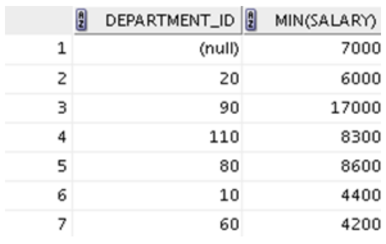
The above query displays the last name, job ID, and salary of all employees whose salary is equal to the minimum salary. The MIN group function returns a single value (2500) to the outer query.

Activity 4: HAVING Clause with Subqueries

We can use subqueries not only in the WHERE clause, but also in the HAVING clause. In this activity, we are going to use subquery in HAVINNG clause. The Oracle server executes the subquery and the results are returned into the HAVING clause of the main query.

```
SELECT    department_id, MIN(salary)
FROM      employees
GROUP BY  department_id
HAVING    MIN(salary) >
                        (SELECT    MIN(salary)
                         FROM      employees
                         WHERE     department_id = 50);
```

Output

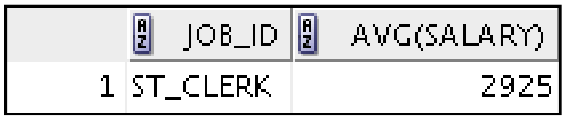


	DEPARTMENT_ID	MIN(SALARY)
1	(null)	7000
2	20	6000
3	90	17000
4	110	8300
5	80	8600
6	10	4400
7	60	4200

The above SQL statement displays all the departments that have a minimum salary greater than that of department 50.

Here's another statement that uses subquery in HAVING clause to find the job with the lowest average salary:

```
SELECT   job_id, AVG(salary)
FROM     employees
GROUP BY job_id
HAVING   AVG(salary) =
                        (SELECT     MIN(AVG(salary))
                         FROM       employees
                         GROUP BY   job_id);
```



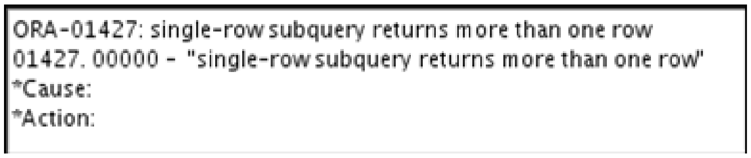
	JOB_ID	AVG(SALARY)
1	ST_CLERK	2925

Activity 5: Single-Row Subquery with more than One Row Returned

A common error with subqueries occurs when more than one row is returned for a single-row subquery. In the following SQL statement, the subquery contains a GROUP BY clause, which implies that the subquery will return multiple rows, one for each group that it finds. In this case, the results of the subquery are 4400, 6000, 2500, 4200, 7000, 17000, and 8300. The outer query takes those results and uses them in its WHERE clause. The WHERE clause contains an equal (=) operator, a single-row comparison operator that expects only one value. The = operator cannot accept more than one value from the subquery and, therefore, generates the error.

```
SELECT    employee_id, last_name
FROM      employees
WHERE     salary =
                  (SELECT     MIN(salary)
                   FROM       employees
                   GROUP BY   department_id);
```

Output



```
ORA-01427: single-row subquery returns more than one row
01427. 00000 - "single-row subquery returns more than one row"
*Cause:
*Action:
```

To correct this error, change the = operator to IN as:

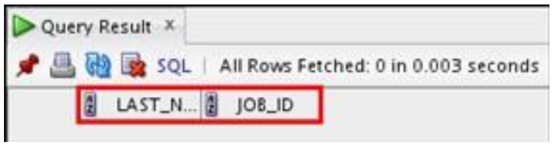
```
SELECT    employee_id, last_name
FROM      employees
WHERE     salary IN
                  (SELECT     MIN(salary)
                   FROM       employees
                   GROUP BY   department_id);
```

Activity 6: No Rows Returned by the Inner Query

Another common problem with subqueries occurs when no rows are returned by the inner query. When this happens, the outer query also returns no rows. Even if a value of null is returned by the subquery, the row is still not returned by the outer query because comparison of two null values yields a null; therefore, the WHERE condition is not true.

```
SELECT    last_name, job_id
FROM      employees
WHERE     job_id =
                  (SELECT     job_id
                   FROM       employees
                   WHERE      last_name = 'Haas');
```

Output



The above statement is correct, but selects no rows when executed because there is no employee named *Haas*. Therefore, the subquery returns no rows.

Activity 7: Using the ANY Operator in Multiple-Row Subqueries

The ANY operator (and its synonym, the SOME operator) compares a value to each value returned by a subquery.

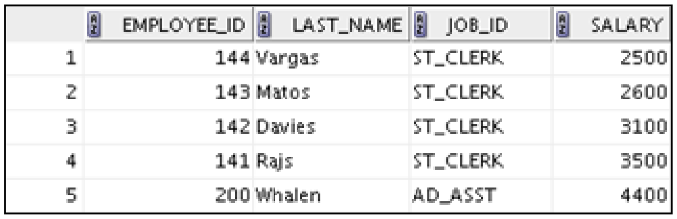
- ***< ANY means less than the maximum.***
- ***> ANY means more than the minimum.***
- ***= ANY is equivalent to IN***

In this activity, we are going to display employees who are not IT programmers and whose salary is less than that of any IT programmer. The maximum salary that a programmer earns is $9,000.

```
SELECT    employee_id, last_name, job_id, salary
FROM      employees
WHERE     salary < ANY
```

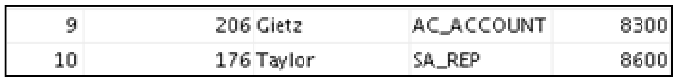
```
                             (SELECT     salary
                             FROM        employees
                             WHERE       job_id = 'IT_PROG')
AND         job_id <> 'IT_PROG';
```

Output



	EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
1	144	Vargas	ST_CLERK	2500
2	143	Matos	ST_CLERK	2600
3	142	Davies	ST_CLERK	3100
4	141	Rajs	ST_CLERK	3500
5	200	Whalen	AD_ASST	4400

…



9	206	Gietz	AC_ACCOUNT	8300
10	176	Taylor	SA_REP	8600

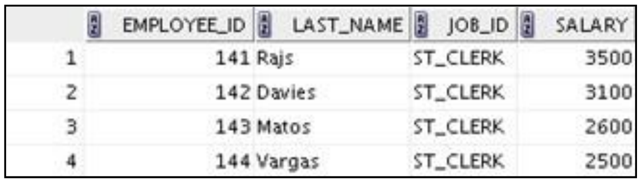
Activity 8: Using the ALL Operator in Multiple-Row Subqueries

The ALL operator compares a value to every value returned by a subquery. > ALL means more than the maximum and < ALL means less than the minimum. The NOT operator can be used with IN, ANY, and ALL operators.

In this activity, we are going to display employees whose salary is less than the salary of all employees with a job ID of IT_PROG and whose job is not IT_PROG.

```
SELECT      employee_id, last_name, job_id, salary
FROM        employees
WHERE       salary < ALL
                             (SELECT     salary
                             FROM        employees
                             WHERE       job_id = 'IT_PROG')
AND         job_id <> 'IT_PROG';
```

Output



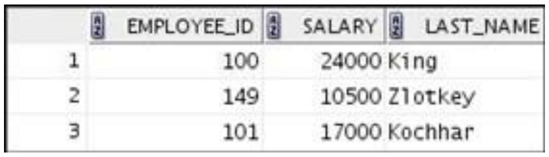
	EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
1	141	Rajs	ST_CLERK	3500
2	142	Davies	ST_CLERK	3100
3	143	Matos	ST_CLERK	2600
4	144	Vargas	ST_CLERK	2500

Activity 9: Using the EXISTS Operator

The EXISTS operator is used in queries where the query result depends on whether or not certain rows exist in a table. It evaluates to TRUE if the subquery returns at least one row. In this activity, we are going to use EXISTS operator to display managers in the EMPLOYEES table who earns a salary more than 10000.

```
SELECT      employee_id, salary, last_name
FROM        employees M
WHERE       EXISTS
                        (SELECT     employee_id
                         FROM       employees W
                         WHERE      (W.manager_id = M.employee_id) AND
                                    W.salary > 10000);
```

Output

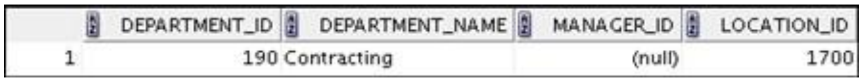


	EMPLOYEE_ID	SALARY	LAST_NAME
1	100	24000	King
2	149	10500	Zlotkey
3	101	17000	Kochhar

In the above statement, for each row in EMPLOYEES table, the condition is checked whether there exists a manager_id who earns a salary more than 10000.

Here's another statement that displays departments that have no employees:

```
SELECT      *
FROM        departments D
WHERE       NOT EXISTS
                        (SELECT     *
                         FROM       employees E
                         WHERE      D.department_id = E.department_id);
```



	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	190	Contracting	(null)	1700

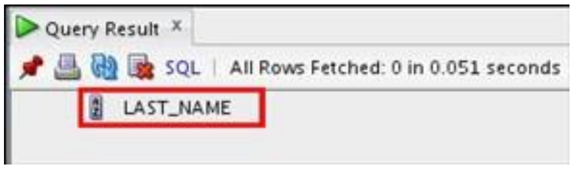
Here, for each row in the DEPARTMENTS table, the condition is checked whether there exists a row in the EMPLOYEES table that has the same department ID. In case no such row exists, the condition is satisfied for the row under consideration and it is selected. If there exists a corresponding row in the EMPLOYEES table, the row is not selected.

Activity 10: Null Values in a Multiple-Row Subquery

The following SQL statement attempts to display all the employees who do not have any subordinates. Logically, this SQL statement should have returned 12 rows. However, the SQL statement does not return any rows. One of the values returned by the inner query is a null value and, therefore, the entire query returns no rows. The reason is that all conditions that compare a null value result in a null. So whenever null values are likely to be part of the results set of a subquery, do not use the NOT IN operator. The NOT IN operator is equivalent to <> ALL

```
SELECT    emp.last_name
FROM      employees emp
WHERE     emp.employee_id NOT IN
                                  (SELECT     mgr.manager_id
                                  FROM        employees mgr);
```

Output



All Rows Fetched: 0 in 0.051 seconds

LAST_NAME

In this case, the correct way is to include a WHERE clause in the subquery to display all employees who do not have any subordinates.

```
SELECT    emp.last_name
FROM      employees emp
WHERE     emp.employee_id NOT IN
                                  (SELECT     mgr.manager_id
                                   FROM       employees mgr
                                   WHERE      mgr.manager_id IS NOT NULL);
```

It is important to note that the null value as part of the results set of a subquery is not a problem if we use the IN operator. The IN operator is equivalent to =ANY.

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Lab Task 1

Write a SQL statement that uses subquery to display the last name and hire date of any employee in the same department as the employee whose last name is provided by the user For example, if the user enters Zlotkey, find all employees who work with Zlotkey (excluding Zlotkey).

Lab Task 2

Write a SQL statement that uses subquery to display the employee number, last name, and salary of all employees who earn more than the average salary. Sort the results in order of ascending salary..

Lab Task 3

Write a SQL statement that uses subquery to display the department number, last name, and job ID for every employee in the Executive department.

Lab Task 4

Write a SQL statement that uses subquery to find those departments where maximum salary is 7000 and above. The employees worked in those departments have already completed one or more jobs. Return all the fields of the departments.

Lab Task 5

Write a SQL statement that uses subquery to find those employees whose department is located in the city 'London'. Return first name, last name, salary, and department ID.

Lab Task 6

Write a SQL statement that uses subquery to display the last name and salary of every employee who reports to King.

Lab 08

Using the Set Operators

Objective:

The objective of this lab is to introduce various set operators that exist in SQL and to make students learn how to write queries by using set operators.

Activity Outcomes:

After completing this lab, students should be able to do the following:

- Describe set operators
- Use a set operator to combine multiple queries into a single query
- Control the order of rows returned

Instructor Note:

Please make sure that you have completed the previous labs.

1) Useful Concepts

Set operators combine the results of two or more component queries into one result. Queries containing set operators are called *compound queries*.

Operator	Returns
UNION	Rows from both queries after eliminating duplications
UNION ALL	Rows from both queries, including all duplications
INTERSECT	Rows that are common to both queries
MINUS	Rows in the first query that are not present in the second query

All set operators have equal precedence. If a SQL statement contains multiple set operators, the Oracle server evaluates them from left (top) to right (bottom) - if no parentheses explicitly specify another order. You should use parentheses to specify the order of evaluation explicitly in queries that use the INTERSECT operator with other set operators.

Rules:

- The expressions in the SELECT lists of the queries must match in number and data type. Queries that use UNION, UNION ALL, INTERSECT, and MINUS operators must have the same number and data type of columns in their SELECT list. The data type of the columns in the SELECT list of the queries in the compound query may not be exactly the same. The column in the second query must be in the same data type group (such as numeric or character) as the corresponding column in the first query.
- Set operators can be used in subqueries.
- You should use parentheses to specify the order of evaluation in queries that use the INTERSECT operator with other set operators. This ensures compliance with emerging SQL standards that will give the INTERSECT operator greater precedence than the other set operators

When a query uses set operators, the Oracle server eliminates duplicate rows automatically except in the case of the UNIONALLoperator. The column names in the output are decided by the column list in the first SELECTstatement. By default, the output is sorted in ascending order of the first column of the SELECT clause.

The corresponding expressions in the SELECT lists of the component queries of a compound query must match in number and data type. If component queries select character data, the data type of the return values is determined as follows:

- If both queries select values of CHAR data type, of equal length, the returned values have the CHAR data type of that length. If the queries select values of CHAR with different lengths, the returned value is VARCHAR2with the length of the larger CHARvalue.
- If either or both of the queries select values of VARCHAR2 data type, the returned values have the VARCHAR2data type.

If component queries select numeric data, the data type of the return values is determined by numeric precedence. If all queries select values of the NUMBERtype, the returned values have the NUMBER data type. In queries using set operators, the Oracle server does not perform implicit conversion across data type groups. Therefore, if the corresponding expressions of component queries resolve to both character data and numeric data, the Oracle server returns an error.

UNION Operator

The UNION operator returns all rows that are selected by either query. Use the UNION operator to return all rows from multiple tables and eliminate any duplicate rows. Guidelines:

- The number of columns being selected must be the same.
- The data types of the columns being selected must be in the same data type group (such as numeric or character).
- The names of the columns need not be identical.
- UNION operates over all of the columns being selected.
- NULL values are not ignored during duplicate checking.
- By default, the output is sorted in ascending order of the columns of the SELECT clause.

UNION ALL operator

The UNION ALL operator returns rows from both queries, including all duplications. The guidelines for UNION and UNION ALL are the same, with the following two exceptions that pertain to UNION ALL: Unlike UNION, duplicate rows are not eliminated and the output is not sorted by default.

INTERSECT Operator

The INTERSECT operator returns rows that are common to both queries.
Guidelines:

- The number of columns and the data types of the columns being selected by the SELECT statements in the queries must be identical in all the SELECT statements used in the query. The names of the columns, however, need not be identical.
- Reversing the order of the intersected tables does not alter the result.
- INTERSECT does not ignore NULL values.

MINUS operator

The MINUS operator returns all the distinct rows selected by the first query, but not present in the second query result set. The number of columns must be the same and the data types of the columns being selected by the SELECT statements in the queries must belong to the same data type group in all the SELECT statements used in the query. The names of the columns, however, need not be identical.

Tables used in this Lab

Two tables are used in this lab: the EMPLOYEES table and the JOB_HISTORY table. You are already familiar with the EMPLOYEES table that stores employee details such as a unique identification number, email address, job identification (such as ST_CLERK, SA_REP, and so on), salary, manager, and so on.

Some of the employees have been with the company for a long time and have switched to different jobs. This is monitored using the JOB_HISTORY table. When an employee switches jobs, the details of the start date and end date of the former job, the job_id (such as ST_CLERK, SA_REP, and so on), and the department are recorded in the JOB_HISTORY table.
There have been instances in the company of people who have held the same position more than once during their tenure with the company. For example, consider the employee Taylor, who joined the company on 24-MAR-2006. Taylor held the job title SA_REP for the period 24- MAR-06 to 31-DEC-06 and the job title SA_MAN for the period 01-JAN-07 to 31-DEC-07. Taylor moved back into the job title of SA_REP, which is his current job title.

2) Solved Lab Activites

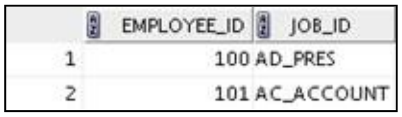
Activity No	*Allocated Time*	*Level of complexity*	*CLO Mapping*
Activity 1	***5 mins***	***Low***	***CLO-5***
Activity 2	***5 mins***	***Low***	***CLO-5***
Activity 3	***5 mins***	***Low***	***CLO-5***
Activity 4	***5 mins***	***Low***	***CLO-5***
Activity 5	***10 mins***	***Medium***	***CLO-5***
Activity 6	***5 mins***	***Low***	***CLO-5***

Activity 1: Using the UNION Operator

In this activity, we are going to display the current and previous job details of all employees while displaying each employee only once.

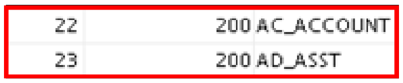
```
SELECT    employee_id, job_id
FROM      employees
UNION
SELECT    employee_id, job_id
FROM      job_history
```

Output



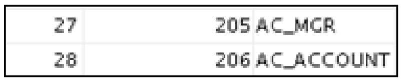
```
     EMPLOYEE_ID  JOB_ID
1            100  AD_PRES
2            101  AC_ACCOUNT
```

…



```
22           200  AC_ACCOUNT
23           200  AD_ASST
```

…

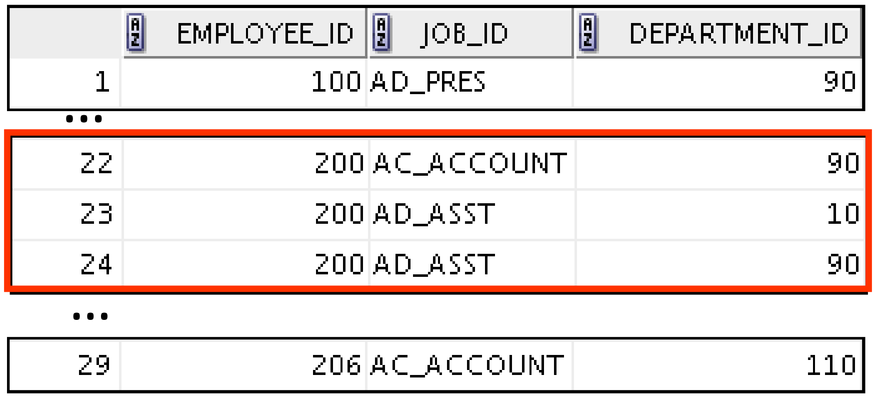


```
27           205  AC_MGR
28           206  AC_ACCOUNT
```

The UNION operator eliminates any duplicate records. If records that occur in both the EMPLOYEES and the JOB_HISTORY tables are identical, the records are displayed only once. We can observe in the output shown above that the record for the employee with the EMPLOYEE_ID 200 appears twice because the JOB_ID is different in each row. Now let's consider the following statement:

```
SELECT    employee_id, job_id, department_id
FROM      employees
UNION
SELECT    employee_id, job_id, department_id
FROM      job_history
```

Here's the output of the above statement and we can see that now employee 200 appears three times. Why? Note the DEPARTMENT_ID values for employee 200. One row has a DEPARTMENT_ID of 90, another 10, and the third 90. Because of these unique combinations of job IDs and department IDs, each row for employee 200 is unique and, therefore, not considered to be a duplicate. Observe that the output is sorted in ascending order of the first column of the SELECT clause (in this case, EMPLOYEE_ID).



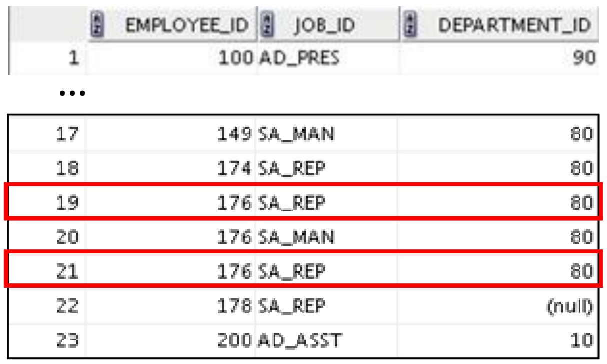
	EMPLOYEE_ID	JOB_ID	DEPARTMENT_ID
1	100	AD_PRES	90
...			
22	200	AC_ACCOUNT	90
23	200	AD_ASST	10
24	200	AD_ASST	90
...			
29	206	AC_ACCOUNT	110

Activity 2: Using the UNION ALL Operator

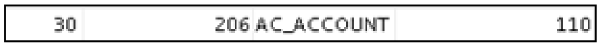
In this activity, we are going to display the current and previous departments of all employees.

```
SELECT    employee_id, job_id, department_id
FROM      employees
UNION ALL
SELECT    employee_id, job_id, department_id
FROM      job_history
ORDER BY  employee_id;
```

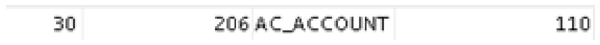
Output



	EMPLOYEE_ID	JOB_ID	DEPARTMENT_ID
1	100	AD_PRES	90
...			
17	149	SA_MAN	80
18	174	SA_REP	80
19	176	SA_REP	80
20	176	SA_MAN	80
21	176	SA_REP	80
22	178	SA_REP	(null)
23	200	AD_ASST	10



```
30        206 AC_ACCOUNT        110
```



```
30        206 AC_ACCOUNT        110
```

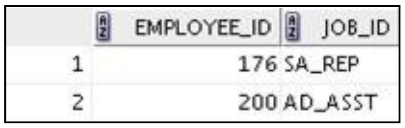
As a result of the execution of the above statement, 30 rows are selected. The combination of the two tables totals to 30 rows. The UNION ALL operator does not eliminate duplicate rows. UNION returns all distinct rows selected by either query. UNION ALL returns all rows selected by either query, including all duplicates.

Activity 3: Using the INTERSECT Operator

In this activity, we are going to display the employee IDs and job IDs of those employees who currently have a job title that is the same as their previous one (that is, they changed jobs but have now gone back to doing the same job they did previously).

```
SELECT    employee_id, job_id
FROM      employees
INTERSECT
SELECT    employee_id, job_id
FROM      job_history;
```

Output



	EMPLOYEE_ID	JOB_ID
1	176	SA_REP
2	200	AD_ASST

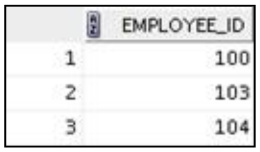
In the above statement, the query returns only those records that have the same values in the selected columns in both tables.

Activity 4: Using the MINUS Operator

In this activity, we are going to display the employee IDs of those employees who have not changed their jobs even once.

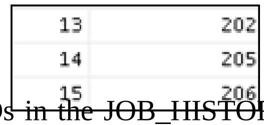
```
SELECT    employee_id
FROM      employees
INTERSECT
SELECT    employee_id
FROM      job_history;
```

Output



	EMPLOYEE_ID
1	100
2	103
3	104

…



13	202
14	205
15	206

In the above statement, the employee IDs in the JOB_HISTORY table are subtracted from those in the EMPLOYEES table. The results set displays the employees remaining after the subtraction; they are represented by rows that exist in the EMPLOYEES table, but do not exist in the JOB_HISTORY table. These are the records of the employees who have not changed their jobs even once.

Activity 5: Matching the SELECT Statements

In this activity, we are going to use the UNION operator to display the location ID, department name, and the state where it is located. Because the expressions in the SELECT lists of the queries must match in number, you can use the dummy columns and the data type conversion functions to comply with this rule.

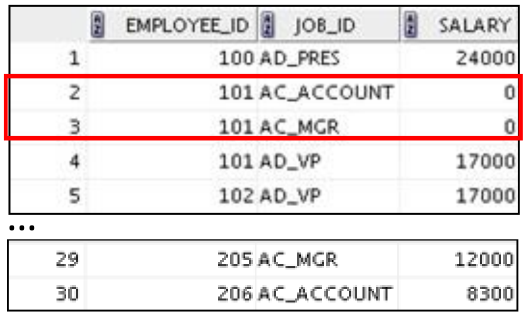
```
SELECT    location_id, department_name "Department",
          TO_CHAR(NULL) "Warehouse location"
FROM      departments
UNION
SELECT    location_id, TO_CHAR(NULL) "Department", state_province
FROM      locations;
```

Output

In the above statement, the column name, *Warehouse location*, is given as the dummy column heading. The *TO_CHAR* function is used in the first query to match the VARCHAR2 data type of the *state_province* column that is retrieved by the second query. Similarly, the *TO_CHAR* function in the second query is used to match the VARCHAR2 data type of the department_name column that is retrieved by the first query.

Here's another statement that displays the employee ID, job ID, and salary of all employees using the UNION operator:

```
SELECT    employee_id, job_id, salary
FROM      employees
UNION
SELECT    employee_id, job_id, 0
FROM      job_history;
```



	EMPLOYEE_ID	JOB_ID	SALARY
1	100	AD_PRES	24000
2	101	AC_ACCOUNT	0
3	101	AC_MGR	0
4	101	AD_VP	17000
5	102	AD_VP	17000
...			
29	205	AC_MGR	12000
30	206	AC_ACCOUNT	8300

The above statement matches the EMPLOYEE_ID and JOB_ID columns in the EMPLOYEES and JOB_HISTORY tables. A literal value of 0 is added to the JOB_HISTORY SELECT statement to match the numeric SALARY column in the EMPLOYEES SELECT statement. In the results shown in the slide, each row in the output that corresponds to a record from the JOB_HISTORY table contains a 0 in the SALARY column.

Activity 6: Using the ORDER BY Clause in Set Operations

The ORDER BY clause can be used only once in a compound query. If used, the ORDER BY clause must be placed at the end of the query. The ORDER BY clause accepts the column name or an alias. By default, the output is sorted in ascending order in the first column of the first SELECT query.

Note: The ORDER BY clause does not recognize the column names of the second SELECT query. To avoid confusion over column names, it is a common practice to ORDER BY column positions.

```
SELECT    employee_id, job_id, salary
FROM      employees
UNION
SELECT    employee_id, job_id, 0
FROM      job_history
ORDER BY  2;
```

Output

In the above statement, if you omit ORDER BY, by default, the output will be sorted in ascending order of employee_id. You cannot use the columns from the second query to sort the output.

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Lab Task 1

Write a SQL statement that that uses set operators to display the ID and name of countries that have no departments located in them.

Lab Task 2

Write a SQL statement that uses set operators to display a list of department IDs for departments that do not contain the job ID ST_CLERK.

Lab Task 3

Write a SQL statement that that uses set operators to display the department ID and department name of employees belongs to Sales and Accounts department.

Lab Task 4

Write a SQL statement that that uses set operators to display the ID and name of departments that have no employees working in them.

Lab Task 5

Write a SQL statement that uses set operators to display the job ID and the department ID for departments 10, 50, and 20, in that order

Lab Task 6

Write a SQL statement that that uses set operators to find all the employees who do not have mangers.

Lab 09

Manipulating Data using DML Statements

Objective:

The objective of this lab is to make student learn how to use the data manipulation language (DML) statements to insert rows into a table, update existing rows in a table, and delete existing rows from a table. The students will also learn how to control transactions with the COMMIT, SAVEPOINT, and ROLLBACK statements.

Activity Outcomes:

After completing this lesson, you should be able to do the following:

- Describe and apply each data manipulation language (DML) statement
- Control transactions

Instructor Note:

Please make sure that you have completed the previous labs.

1) Useful Concepts

Data manipulation language (DML) is a core part of SQL. When you want to add, update, or delete data in the database, you execute a DML statement. A collection of DML statements that form a logical unit of work is called a transaction.

Consider a banking database. When a bank customer transfers money from a savings account to a checking account, the transaction might consist of three separate operations: decreasing the savings account, increasing the checking account, and recording the transaction in the transaction journal. The Oracle server must guarantee that all the three SQL statements are performed to maintain the accounts in proper balance. When something prevents one of the statements in the transaction from executing, the other statements of the transaction must be undone.

Note: Most of the DML statements in this lesson assume that no constraints on the table are violated. Constraints are discussed later in the next lab. In SQL Developer, click the Run Script icon or press [F5] to run the DML statements. The feedback messages will be shown on the Script Output tabbed page.

INSERT Statement

```
INSERT INTO table [(column [, column...])]
VALUES      (value [, value...]);
```

We can add new rows to a table by issuing the INSERT statement. In the above syntax:

- *table* is the name of the table
- *column* is the name of the column in the table to populate
- *value* is the corresponding value for the column

Note: This statement with the VALUES clause adds only one row at a time to a table.

The INSERT statement can also be used to add rows to a table where the values are derived from existing tables. To do that, in place of the VALUES clause, you use a subquery as in the following syntax:

```
INSERT INTO table [(column [, column...])]
VALUES      subquery;
```

The number of columns and their data types in the column list of the INSERT clause must match the number of values and their data types in the subquery. Zero or more rows are added depending on the number of rows returned by the subquery.

While insertion, the Oracle server automatically enforces all data types, data ranges, and data integrity constraints. Any column that is not listed explicitly obtains a null value in the new row unless we have default values for the missing columns that are used. Common errors that can occur during user input are checked in the following order:

- Mandatory value missing for a NOT NULL column
- Duplicate value violating any unique or primary key constraint
- Any value violating a CHECK constraint
- Referential integrity maintained for foreign key constraint
- Data type mismatches or values too wide to fit in column

UPDATE Statement

We can modify the existing values in a table by using the UPDATE statement.

```
UPDATE      table
SET         column = value [, column = value, ...]
[WHERE      condition];
```

In the above syntax:

- *table* is the name of the table
- *column* is the name of the column in the table to populate
- *value* is the corresponding value or subquery for the column
- *condition* identifies the rows to be updated and is composed of column names, expressions, constants, subqueries, and comparison operators

Note: In general, use the primary key column in the WHERE clause to identify a single row for update. Using other columns can unexpectedly cause several rows to be updated. For example, identifying a single row in the EMPLOYEES table by name is dangerous, because more than one employee may have the same name.

DELETE Statement

```
DELETE [FROM]     table
[WHERE            condition];
```

We can remove existing rows from a table by using the DELETEstatement. In the above syntax:

- *table* is the name of the table
- *condition* identifies the rows to be deleted, and is composed of column names, expressions, constants, subqueries, and comparison operators

Note: If no rows are deleted, the message “0 rows deleted” is returned (on the Script Output tab in SQL Developer).

2) Solved Lab Activites

Activity No	*Allocated Time*	*Level of complexity*	*CLO Mapping*
Activity 1	***5 mins***	***Low***	***CLO-5***
Activity 2	***5 mins***	***Low***	***CLO-5***
Activity 3	***5 mins***	***Low***	***CLO-5***
Activity 4	***5 mins***	***Low***	***CLO-5***
Activity 5	***5 mins***	***Low***	***CLO-5***
Activity 6	***5 mins***	***Low***	***CLO-5***
Activity 7	***5 mins***	***Low***	***CLO-5***

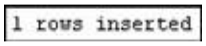
Activity 8	***5 mins***	***Low***	***CLO-5***
Activity 9	***5 mins***	***Low***	***CLO-5***
Activity 10	***5 mins***	***Low***	***CLO-5***
Activity 11	***5 mins***	***Low***	***CLO-5***
Activity 12	***5 mins***	***Low***	***CLO-5***

Activity 1: Inserting New Rows

In this lab activity, we are going to demonstrate the use of INSERT statement by inserting a row in departments table. Because you can insert a new row that contains values for each column, the column list is not required in the INSERT clause. However, if you do not use the column list, the values must be listed according to the default order of the columns in the table, and a value must be provided for each column. For clarity, it's better use the column list in the INSERT clause. Normally, we enclose character and date values within single quotation marks; however, it is not recommended that you enclose numeric values within single quotation marks.

```
INSERT INTO departments (department_id, department_name, manager_id,
                         location_id)
VALUES (70, 'Public Relations', 100, 1700);
```

Output



```
1 rows inserted
```

Activity 2: Inserting Rows with Null Values

While inserting rows having null values, we can use either of the two methods listed in the following table:

Method	Description
Implicit	Omit the column from the column list.
Explicit	Specify the NULL keyword in the VALUES list; specify the empty string ('') in the VALUES list for character strings and dates.

In this lab activity, we are going to insert rows in department table using both the methods.

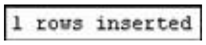
Implicit method: Omit the column from the column list.

```
INSERT INTO      departments (department_id, department_name)
VALUES           (30, 'Purchasing');
```

Explicit method: Specify the NULL keyword in the VALUES clause.

```
INSERT INTO       departments
VALUES            (100, 'Finance', NULL, NULL);
```

Output

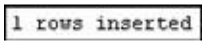

```
1 rows inserted
```

Activity 3: Inserting Special Values

In this lab activity, we are going to demonstrate the use of functions to enter special values in the employees table. The following statement records information for employee Popp in the EMPLOYEES table. It supplies the current date and time in the HIRE_DATE column. It uses the SYSDATE function that returns the current date and time of the database server. You may also use the CURRENT_DATE function to get the current date in the session time zone. You can also use the USER function when inserting rows in a table. The USER function records the current username.

```
INSERT INTO employees (employee_id, first_name, last_name, email,
                       phone_number, hire_date, job_id, salary,
                       commission_pct, manager_id, department_id)
VALUES                 (113, 'Louis', 'Popp', 'LPOPP',
                        '515.124.4567',SYSDATE, 'AC_ACCOUNT', 6900, NULL,
                        205, 110);
```

Output


```
1 rows inserted
```

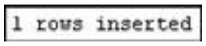
Activity 4: Inserting Specific Date and Time Values

The DD-MON-RR format is generally used to insert a date value. With the RR format, the system provides the correct century automatically. You may also supply the date value in the DD-MON-YYYY format. This is recommended because it clearly specifies the century and does not depend on the internal RR format logic of specifying the correct century. If a date must be entered in a format other than the default format (for example, with another century or a specific time), you must use the TO_DATE function.

In this lab activity, we are going to record information for employee Raphealy in the EMPLOYEES table. It sets the HIRE_DATE column to be February 3, 2003.

```
INSERT INTO       employees
VALUES            (114, 'Den', 'Raphealy', 'DRAPHEAL', '515.127.4561',
                   TO_DATE('FEB 3, 2003', 'MON DD, YYYY'), 'SA_REP', 11000,
                   0.2, 100, 60);
```

Output



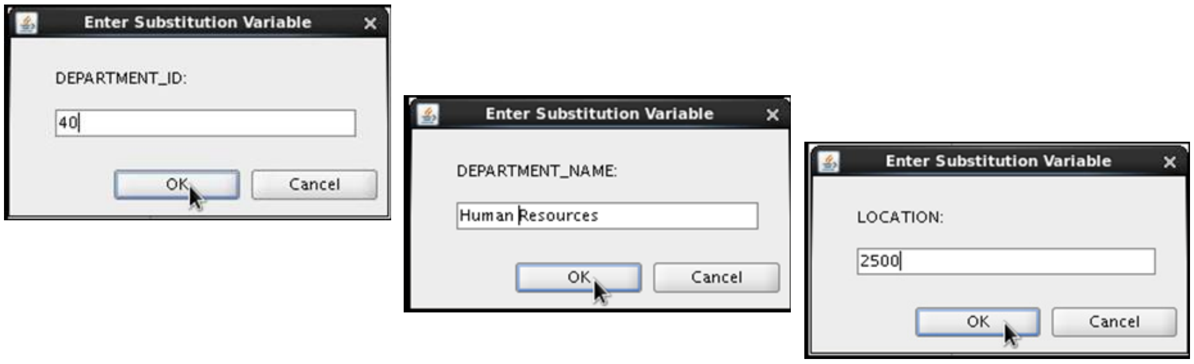
```
1 rows inserted
```

Activity 5: Creating a Script

We can also save commands with substitution variables to a file and execute the commands in the file. When the script file is run, you are prompted for input for each of the ampersand (&) substitution variables. After entering a value for the substitution variable, click the OK button. The values that you input are then substituted into the statement. This enables you to run the same script file over and over, but supply a different set of values each time you run it.

In this lab activity, we are going to create a script with substitution variables to record information for a department in the DEPARTMENTS table.

```
INSERT INTO departments (department_id, department_name, location_id)
VALUES      (&department_id, '&department_name',&location);
```



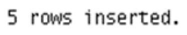
Output

Activity 6: Copying Rows from Another Table

In this activity, we are going to use a subquery to fetch the data which in turn will be inserted into sales_reps table. (Here we assume that we have already created the sales_reps table using the CREATE TABLE statement which will be covered in the next lab)

```
INSERT INTO sales_reps (id, name, salary, commission_pct)
SELECT      employee_id, last_name, salary, commission_pct
FROM        employees
WHERE       job_id LIKE '%REP%';
```

Output



```
5 rows inserted.
```

Here’s another statement that creates a copy of the rows of employees table by using SELECT * in the subquery:

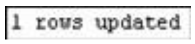
```
INSERT INTO copy_emp
SELECT      *
FROM        employees;
```

Activity 7: Updating Rows in a Tables

The UPDATE statement modifies the values of a specific row or rows if the WHERE clause is specified. In this activity, we are going to use the UPDATE statement to transfer the employee 113 (Popp) to department 50. If you omit the WHERE clause, values for all the rows in the table are modified.

```
UPDATE    employees
SET       department_id = 50
WHERE     employee_id = 113;
```

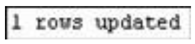
Output



```
1 rows updated
```

Here’s another UPDATE statement to update an employee who was a SA_REP and has now changed his job to an IT_PROG. Consequently, his JOB_IDneeds to be updated and the commission field needs to be set to NULL.

```
UPDATE    employees
SET       job_id = 'IT_PROG', commission_pct = NULL
WHERE     employee_id = 114;
```



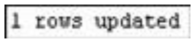
```
1 rows updated
```

Activity 8: Updating Two Columns with a Subquery

In this activity, we are going to update multiple columns in the SET clause of an UPDATE statement by writing multiple subqueries.

```
UPDATE      employees
SET         (job_id,salary) = (SELECT     job_id,salary
                               FROM       employees
                               WHERE      employee_id = 205)
WHERE       employee_id = 103;
```

Output

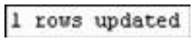

1 rows updated

Activity 9: Updating Rows Based on Another Table

In this lab activity, we are going to use the subqueries in the UPDATE statements to update values in the copy_emp table based on the values from the EMPLOYEES table. It changes the department number of all employees with employee 200's job ID to employee 100's current department number.

```
UPDATE      copy_emp
SET         department_id = (SELECT      department_id
                             FROM        employees
                             WHERE       employee_id = 100)
WHERE       employee_id =   (SELECT      job_id
                             FROM        employees
                             WHERE       employee_id = 200);
```

Output

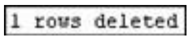

1 rows updated

Activity 10: Deleting a Row from a Table

In this activity, we are going to use DELETE statement to delete the Finance department from the DEPARTMENTS table.

```
DELETE FROM departments
WHERE       department_name = 'Finance';
```

Output


1 rows deleted

However, if you omit the WHERE clause, all rows in the table are deleted. In the following statement, all rows from the COPY_EMP table would get deleted, because no WHERE clause was specified.

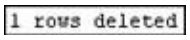
```
DELETE FROM copy_emp;
```

Activity 11: Deleting Rows Based on Another Table

In the activity, we are going to use subqueries to delete rows from a table based on values from another table. Specifically, we are going to delete all the employees in a department, where the department name contains the string Public.

```
DELETE FROM employees
WHERE       department_id IN
                               (SELECT     department_id
                                FROM       departments
                                WHERE      department_name LIKE '%Public%');
```

Output



```
1 rows deleted
```

In the above statement, the subquery searches the DEPARTMENTS table to find the department number based on the department name containing the string Public. The subquery then feeds the department number to the main query, which deletes rows of data from the EMPLOYEES table based on this department number.

Activity 12: Efficient method of Emptying a Table

A more efficient method of emptying a table is by using the TRUNCATE statement.

You can use the TRUNCATE statement to quickly remove all rows from a table or cluster. Removing rows with the TRUNCATE statement is faster than removing them with the DELETE statement for the following reasons:

- ***The TRUNCATE statement is a data definition language (DDL) statement and generates no rollback information. Rollback information is covered later in this lab.***
- ***Truncating a table does not fire the delete triggers of the table.***

If the table is the parent of a referential integrity constraint, you cannot truncate the table. You need to disable the constraint before issuing the TRUNCATE statement. Disabling constraints is covered in the next lab.

```
TRUNCATE TABLE copy_emp;
```

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Lab Task 1

Write a SQL statement that increases the commission percentage for every employee in department 80 by 5%.

Lab Task 2

Write a SQL statement that inserts you as an employee.

Lab Task 3

Write a SQL statement that inserts Network Operations department located in Africa.

Lab Task 4

Write a SQL statement that updates the Sales department's location to London

Lab Task 5

Write a SQL statement that deletes all the employees who works for Steven Kings

Lab Task 6

Write a SQL statement that deletes you as an employee.

Lab 10

Using DDL Statements to Create and Manage Tables

Objective:

The objective of this lab is to introduce data definition language (DDL) statements. It will make students learn the basics of how to create simple tables, alter them, and remove them. The data types available in DDL will be shown and schema concepts will be introduced. Constraints will be discussed and the exception messages that are generated from violating constraints during DML operations will be shown and explained.

Activity Outcomes:

After completing this lesson, you should be able to do the following:

- Categorize the main database objects
- Review the table structure
- List the data types that are available for columns
- Create a simple table
- Explain how constraints are created at the time of table creation
- Describe how schema objects work

Instructor Note:

Please make sure that you have completed the previous labs.

1) Useful Concepts

The Oracle database can contain multiple data structures. Each structure should be outlined in the database design so that it can be created during the build stage of database development.

- Table: Stores data
- View: Is a subset of data from one or more table
- Sequence: Generates numeric values
- Index: Improves the performance of some queries
- Synonym: Gives alternative name to an object

Oracle Table Structures

- Tables can be created at any time, even when users are using the database.
- You do not need to specify the size of a table. The size is ultimately defined by the amount of space allocated to the database as a whole. It is important, however, to estimate how much space a table will use over time.
- Table structure can be modified online.

Note: More database objects are available, but are not covered in this course.

Naming Rules

- You name database tables and columns according to the standard rules for naming any Oracle database object.
- Table names and column names must begin with a letter and be 1–30 characters long.
- Names must contain only the characters A–Z, a–z, 0–9, _ (underscore), $, and # (legal characters, but their use is discouraged).
- Names must not duplicate the name of another object owned by the same Oracle server user.
- Names must not be an Oracle server–reserved word.
    - You may also use quoted identifiers to represent the name of an object. A quoted identifier begins and ends with double quotation marks (“”). If you name a schema object using a quoted identifier, you must use the double quotation marks whenever you refer to that object. Quoted identifiers can be reserved words, although this is not recommended.

Naming Guidelines

- Use descriptive names for tables and other database objects.
- Names are not case-sensitive. For example, EMPLOYEES is treated to be the same name as eMPloyees or eMpLOYEES. However, quoted identifiers are case-sensitive.

Data Types

When you identify a column for a table, you need to provide a data type for the column. There are several data types available:

Data Type	Description
VARCHAR2(*size*)	Variable-length character data (A maximum *size* must be specified: minimum *size* is 1.) Maximum size is: • 32767 bytes if MAX_SQL_STRING_SIZE = EXTENDED • 4000 bytes if MAX_SQL_STRING_SIZE = LEGACY
CHAR [(*size*)]	Fixed-length character data of length *size* bytes (Default and minimum *size* is 1; maximum *size* is 2,000.)
NUMBER [(*p*,*s*)]	Number having precision *p* and scale *s* (Precision is the total number of decimal digits and scale is the number of digits to the right of the decimal point; precision can range from 1 to 38, and scale can range from –84 to 127.)
DATE	Date and time values to the nearest second between January 1, 4712 B.C., and December 31, 9999 A.D.
LONG	Variable-length character data (up to 2 GB)
CLOB	A character large object containing single-byte or multibyte characters. Maximum size is (4 gigabytes - 1) * (DB_BLOCK_SIZE); stores national character set data.
NCLOB	A character large object containing Unicode characters. Both fixed-width and variable-width character sets are supported, both using the database national character set. Maximum size is (4 gigabytes - 1) * (database block size); stores national character set data.
RAW(size)	Raw binary data of length *size* bytes. You must specify *size* for a RAWvalue. Maximum *size* is: 32767 bytes if MAX_SQL_STRING_SIZE = EXTENDED 4000 bytes if MAX_SQL_STRING_SIZE = LEGACY
LONG RAW	Raw binary data of variable length up to 2 gigabytes
BLOB	A binary large object. Maximum size is (4 gigabytes - 1) * (DB_BLOCK_SIZE initialization parameter (8 TB to 128 TB)).
BFILE	Binary data stored in an external file (up to 4 GB)
ROWID	Base 64 string representing the unique address of a row in its table. This data type is primarily for values returned by the ROWID pseudocolumn

DEFAULT Option

When you define a table, you can specify that a column should be given a default value by using the DEFAULT option. This option prevents null values from entering the columns when a row is inserted without a value for the column. The default value can be a literal, an expression, or a SQL function (such as SYSDATE or USER), but the value cannot be the name of another column or a pseudocolumn (such as NEXTVAL or CURRVAL). The default expression must match the data type of the column.
Consider the following examples:

INSERT INTO hire_dates values(45, NULL);

The preceding statement will insert the null value rather than the default value.

INSERT INTO hire_dates(id) values(35);

The preceding statement will insert SYSDATE for the HIRE_DATE column.

CREATE TABLE Statement

You create tables to store data by executing the SQL CREATE TABLE statement. This statement is one of the DDL statements that are a subset of the SQL statements used to create, modify, or remove Oracle Database structures. These statements have an immediate effect on the database and they also record information in the data dictionary.
To create a table, a user must have the CREATE TABLE privilege and a storage area in which to create objects. The database administrator (DBA) uses data control language (DCL) statements to grant privileges to users.

```
CREATE TABLE [schema.]table (column datatype [DEFAULT expr][, ...]);
```

In the above syntax:

- *schema* is the same as the owner's name
- *table* is the name of the table
- *DEFAULT expr* specifies a default value if a value is omitted in the INSERT
- statement
- *column* is the name of the column
- *datatype* is the column's data type and length

Note: The CREATE ANY TABLE privilege is needed to create a table in any schema other than the user's schema.

Overview of Constraints

The Oracle server uses constraints to prevent invalid data entry into tables. You can use constraints to do the following:

- Enforce rules on the data in a table whenever a row is inserted, updated, or deleted from that table. The constraint must be satisfied for the operation to succeed.
- Prevent the dropping of a table if there are dependencies from other tables.
- Provide rules for Oracle tools, such as Oracle Developer.

Data Integrity Constraints

Constraint	Description
NOT NULL	Specifies that the column cannot contain a null value
UNIQUE	Specifies a column or combination of columns whose values must be unique for all rows in the table
PRIMARYKEY	Uniquely identifies each row of the table
FOREIGNKEY	Establishes and enforces a referential integrity between the column and a column of the referenced table such that values in one table match values in another table.
CHECK	Specifies a condition that must be true

Constraints are easy to reference if you give them a meaningful name. Constraint names must follow the standard object-naming rules, except that the name cannot be the same as another object owned by the same user. If you do not name your constraint, the Oracle server generates a name with the format SYS_Cn, where n is an integer so that the constraint name is unique.

Constraints can be defined at the time of table creation or after the creation of the table. You can define a constraint at the column or table level. Functionally, a table-level constraint is the same as a column-level constraint.

```
CREATE TABLE [schema.]table
             (column datatype [DEFAULT expr] [column_constraint],
             ...
             [table_constraint][,...]);
```

Defining Constraints

You can create constraints at either the column level or table level. Constraints defined at the column level are included when the column is defined. Table-level constraints are defined at the end of the table definition, and must refer to the column or columns on which the constraint pertains in a set of parentheses. It is mainly the syntax that differentiates the two; otherwise, functionally, a column-level constraint is the same as a table-level constraint.
NOT NULL constraints must be defined at the column level.
Constraints that apply to more than one column must be defined at the table level.
In the above syntax:

- *schema* is the same as the owner's name
- *table* is the name of the table
- *DEFAULT expr* specifies a default value to be used if a value is omitted in the INSERT statement
- *column* is the name of the column
- *datatype* is the column's data type and length
- *column_constraint* is an integrity constraint as part of the column definition

- *table_constraint* is an integrity constraint as part of the table definition

ALTER TABLE Statement

ALTER TABLE statement can be used to add columns to a table, modify columns, and drop columns from a table.

```
ALTER TABLE table
ADD         (column datatype [DEFAULT expr]
            [, column datatype]...);
```

```
ALTER TABLE table
MODIFY      (column datatype [DEFAULT expr]
            [, column datatype]...);
```

```
ALTER TABLE table
DROP        (column [, column] …);
```

In the above syntax:

- *table* is the name of the table
- *ADD|MODIFY|DROP* is the type of modification
- *column* is the name of the column
- *datatype* is the data type and length of the column
- *DEFAULT expr* specifies the default value for a column

DROP TABLE Statement

```
DROP TABLE table [PURGE]
```

The DROP TABLE statement moves a table to the recycle bin or removes the table and all its data from the database entirely. Unless you specify the PURGE clause, the DROP TABLE statement does not result in space being released back to the tablespace for use by other objects, and the space continues to count toward the user's space quota. Dropping a table invalidates the dependent objects and removes object privileges on the table.
When you drop a table, the database loses all the data in the table and all the indexes associated with it.

2) Solved Lab Activites

Activity No	*Allocated Time*	*Level of complexity*	*CLO Mapping*
Activity 1	***5 mins***	***Low***	***CLO-5***
Activity 2	***5 mins***	***Low***	***CLO-5***
Activity 3	***5 mins***	***Low***	***CLO-5***
Activity 4	***5 mins***	***Low***	***CLO-5***

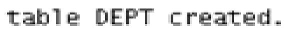
Activity 5	***10 mins***	***Medium***	***CLO-5***
Activity 6	***5 mins***	***Low***	***CLO-5***
Activity 7	***5 mins***	***Low***	***CLO-5***
Activity 8	***5 mins***	***Low***	***CLO-5***
Activity 9	***5 mins***	***Low***	***CLO-5***

Activity 1: Creating a Table

In this activity, we are going to create the DEPT table with four columns: DEPTNO, DNAME, LOC, and CREATE_DATE. The CREATE_DATE column has a default value. If a value is not provided for an INSERT statement, the system date is automatically inserted.

```
CREATE TABLE      dept
                  (deptno     NUMBER(2),
                  Dname       VARCHAR2(14),
                  Loc         VARCHAR2(13),
                  create_date DATE DEFAULT SYSDATE);
```

Output



```
table DEPT created.
```

To confirm that the table was created, run the DESCRIBE command. Because creating a table is a DDL statement, an automatic commit takes place when this statement is executed.

Activity 2: Defining Constraints

In this activity, we are going to use both the column-level syntax and the table-level syntax to define a primary key constraint on the EMPLOYEE_ID column of the EMPLOYEE table.

```
CREATE TABLE      employee
                  (employee_id      NUMBER(6)
                  CONSTRAINT        emp_emp_id_pk PRIMARY KEY,
                  first_name        VARCHAR2(20),
                  ...);
```

```
CREATE TABLE      employee
                  (employee_id      NUMBER(6),
                  first_name        VARCHAR2(20),
                  ...
                  Job_id            VARCHAR2(10) NOT NULL,
                  CONSTRAINT        emp_emp_id_pk PRIMARY KEY(employee_id)
);
```

Activity 3: Defining Foreign Key Constraint

In this activity, we are going to define DEPARTMENT_ID as the foreign key in the EMPLOYEE table (dependent or child table); it references the DEPARTMENT_ID column of the DEPARTMENTS table (the referenced or parent table).

```
CREATE TABLE employee
                 (employee_id      NUMBER(6),
                  last_name        VARCHAR2(25) NOT NULL,
                  email            VARCHAR2(25),
                  salary           NUMBER(8,2),
                  commission_pct   NUMBER(2,2),
                  hire_date        DATE NOT NULL,
                  ...
                  department_id    NUMBER(4),
                  CONSTRAINT       emp_dept_fk FOREIGN KEY(department_id)
                                   REFERENCES departments(department_id)
                 );
```

The foreign key (or referential integrity) constraint is defined in the child table and the table containing the referenced column is the parent table. The foreign key is defined using a combination of the following keywords:

- FOREIGN KEY is used to define the column in the child table at the table-constraint level.
- REFERENCES identifies the table and the column in the parent table.
- ON DELETE CASCADE indicates that when a row in the parent table is deleted, the dependent rows in the child table are also deleted.
- ON DELETE SET NULL indicates that when a row in the parent table is deleted, the foreign key values are set to null.

The default behavior is called the restrict rule, which disallows the update or deletion of referenced data.

Without the ON DELETE CASCADE or the ON DELETE SET NULL options, the row in the parent table cannot be deleted if it is referenced in the child table. And these keywords cannot be used in column-level syntax.

Activity 4: Defining Check Constraint

In this activity, we are going to apply check constraint on the salary attribute of the employees table.

```
CREATE TABLE employee
                       (...
                       salary NUMBER(8,2) CONSTRAINT emp_salary_min
                       CHECK (salary > 0),
                       ...);
```

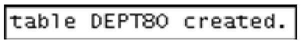
The CHECK constraint defines a condition that each row must satisfy. The condition can use the same constructs as the query conditions. A single column can have multiple CHECK constraints that refer to the column in its definition. There is no limit to the number of CHECK constraints that you can define on a column. CHECK constraints can be defined at the column level or table level.

Activity 5: Creating a Table Using a Subquery

A second method for creating a table is to apply the AS subquery clause, which both creates the table and inserts rows returned from the subquery. In this activity, we are going to create a table named DEPT80, which contains details of all the employees working in department 80. Notice that the data for the DEPT80 table comes from the EMPLOYEES table.

```
CREATE TABLE dept80 AS
                  SELECT      employee_id, last_name, salary*12 ANNSAL,
                              hire_date
                  FROM        employees
                  WHERE       department_id = 80;
```

Output



```
table DEPT80 created.
```

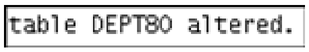
You can verify the existence of a database table and check the column definitions by using the DESCRIBE command.

Activity 6: Adding a column

In this activity, we are going to add a column named JOB_ID to the DEPT80 table. The JOB_ID column would become the last column in the table.

```
ALTER TABLE dept80
ADD         (job_id VARCHAR2(9));
```

Output



```
table DEPT80 altered.
```

If a table already contains rows when a column is added, the new column is initially null or takes the default value for all the rows. You can add a mandatory NOT NULL column to a table that contains data in the other columns only if you specify a default value. You can add a NOT NULL column to an empty table without the default value.

Activity 7: Modifying a Column

We can modify a column definition by using the ALTER TABLE statement with the MODIFY clause. Column modification can include changes to a column's data type, size, and default value. In this

activity, we are going to modify a column named last_name of the DEPT80 table by increasing the width of the VARCHAR2 from 25 to 30.

```
ALTER TABLE dept80
MODIFY      (last_name VARCHAR2(30));
```

Output

```
table DEPT80 altered.
```

Activity 8: Dropping a column

In this activity, we are going to drop a column named JOB_ID from the DEPT80 table.

```
ALTER TABLE dept80
DROP        (job_id);
```

Output

```
table DEPT80 altered.
```

Activity 9: Dropping a table

In this activity, we are going to drop the DEPT80 table using the DROP TABLE statement.

```
DROP TABLE dept80;
```

Output

```
table DEPT80 dropped.
```

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Lab Task 1

Write a SQL statement that creates and populate *the DEPT2 table with data from the DEPARTMENTS table. Include only columns that you need.*

Lab Task 2

Write a SQL statement that creates the EMPLOYEES2 table based on the structure of the EMPLOYEES table. Include only the EMPLOYEE_ID, FIRST_NAME, LAST_NAME, SALARY, and DEPARTMENT_ID *columns. Name the columns in your new table ID, FIRST_NAME, LAST_NAME, SALARY, and DEPT_ID, respectively.*

Lab Task 3

Write a SQL statement that inserts one row in Employees2. Drop table Employees2

Lab Task 4

Write a SQL statement that alters the data type of first_name to varchar2(50)

Lab 11

Indexes, Views, and Data Control Language

Objective:

The objective of this lab is to introduce the views and indexes first and that how they can be created. Later on, the Data Control Language (DCL) will be introduced which is used for controlling the access to the table/views and hence securing the database.

Activity Outcomes:

After completing this lesson, you should be able to do the following:

- Create and manage views
- Create indexes
- Use DCL commands Grant and Revoke for database authorization

Instructor Note:

Please make sure that you have completed the previous labs.

1) Useful Concepts

Index

An index is a schema object that contains an entry for each value that appears in the indexed column(s) of the table or cluster and provides direct, fast access to rows. Oracle Database supports several types of indices:

- Normal indexes. (By default, Oracle Database creates B-tree indexes.)
- Bitmap indexes, which store ROWIDS associated with a key value as a bitmap
- Function-based indexes, which are based on expressions. They enable you to construct queries that evaluate the value returned by an expression, which in turn may include built-in or user-defined functions.

Views

An Oracle VIEW is a virtual table that does not physically exist. Rather, it corresponds to a query possibly joining one or more tables. A view once created can be referred in an SQL statement in the same way as a table. Whenever a view is referred, the query to which it corresponds to is executed and the resultant table is generated which in turn in used.

Data Control Language (DCL)

The DCL language is used for controlling the access to the table and hence securing the database. DCL is used to provide certain privileges to a particular user. Privileges are rights to be allocated. The privilege commands are *Grant* and *Revoke*.

GRANT Statement

You can use the GRANTstatement to:

- Assign privileges to a specific user or role, or to all users, to perform actions on database objects
- Grant a role to a user, to PUBLIC, or to another role

You can grant privileges on an object if you are the owner of the database. You can grant privileges to all users by using the PUBLIC keyword. When PUBLIC is specified, the privileges or roles affect all current and future users.

Oracle Database provides a variety of privilege types to grant privileges to a user or role:

- Use the ALL PRIVILEGES privilege type to grant all privileges to the user or role for the specified table.
- Use the DELETE privilege type to grant permission to delete rows from the specified table.
- Use the INSERT privilege type to grant permission to insert rows into the specified table.
- Use the REFERENCES privilege type to grant permission to create a foreign key reference to the specified table.

- Use the SELECT privilege type to grant permission to perform SELECT statements on a table or view.
- Use the UPDATE privilege type to grant permission to use the UPDATE statement on the specified table.

REVOKE Statement

The REVOKE statement removes privileges from a specific user (or users) or role to perform actions on database objects. It performs the following operations:

- Revokes a role from a user, from PUBLIC, or from another role
- Revokes privileges for an object if you are the owner of the object or the database owner

Note: To revoke a role or system privilege, you must have been granted the privilege with the ADMIN OPTION

2) Solved Lab Tasks

Activity No	*Allocated Time*	*Level of complexity*	*CLO Mapping*
Activity 1	***10 mins***	***Medium***	***CLO-5***
Activity 2	***5 mins***	***Low***	***CLO-5***
Activity 3	***5 mins***	***Low***	***CLO-5***
Activity 4	***10 mins***	***Medium***	***CLO-5***
Activity 5	***5 mins***	***Low***	***CLO-5***
Activity 6	***5 mins***	***Low***	***CLO-5***
Activity 7	***5 mins***	***Low***	***CLO-5***
Activity 8	***5 mins***	***Low***	***CLO-5***

Activity 1: Creating Indexes

In this activity, we are going to demonstrate the creation of various types of indexes.

The following statement creates a normal index on a single attribute:

```
CREATE INDEX employee_index ON Employees( employee_id);
```

The following statement creates a normal index on a single column which contains unique values:

```
CREATE UNIQUE INDEX employee_emailindex ON Employees(email);
```

The following statement creates a normal index on a more than one attribute:

```
CREATE INDEX employee_nameIndex ON Employees(first_name, last_name);
```

Suppose we have gender column in our employees table, and it has two distinct values, *F* for female and *M* for male. When a column has a few distinct values, we say that this column has low cardinality. Oracle has a special kind of index for these types of columns which is called a bitmap index. The following statement creates a bitmap index on the gender column of employees table:

```
CREATE BITMAP INDEX gender_index  ON employees(gender);
```

Function based indexes can be created using single row function or using any expression. The following statement creates a function-based index on the first_name column of employees table:

```
CREATE INDEX firstNameIndex ON employees (UPPER(first_name));
```

Activity 2: Changing Index

In this activity, we are going to change the index name of an already built index.

```
ALTER INDEX employee_nameIndex RENAME TO full_nameIndex;
```

Activity 3: Dropping Indexes

In this activity, we are going to drop an already built index.

```
DROP INDEX employee_nameIndex;
```

Activity 4: Creating Views

In this activity, we are going to demonstrate the creation of views.

The following statement creates a horizontal view in which we are filtering rows:

```
CREATE VIEW lowSalaryEmp AS
          SELECT  *
          FROM    employees
          WHERE   salary < 20000;
```

The following statement creates a vertical view in which we are projecting selected columns:

```
CREATE VIEW allEmp AS
          SELECT  first_name, last_name
          FROM    employees;
```

The following statement creates a view which is a combination of horizontal and vertical view:

```
CREATE VIEW lowSalEmpList AS
          SELECT  first_name, last_name
          FROM    employees
          WHERE   salary < 20000;
```

The following statement creates a view which is a join of two tables:

```
CREATE VIEW deptEmployee AS
         SELECT  departments.department_name, employees.first_name
         FROM    departments INNER JOIN employees
         ON      departments.department_id = employees.department_id;
```

Note: The select statement may contain single line or aggregate functions in queries.

Activity 5: Using and Changing Views

Views can be used as normal tables used in SELECT statements. Views can be changed or re-defined using REPLACE command. In this activity, we are going to demonstrate their usage in the SELECT statements as well as how they can be replaced with another query.

```
SELECT   first_name
FROM     allEmp;
```

```
REPLACE VIEW lowSalEmpList AS
         SELECT  first_name, last_name
         FROM    employees
         WHERE   salary < 30000;
```

Activity 6: Dropping Views

By using the DROP VIEW command, we can remove the view from the database.

```
DROP VIEW allEmp;
```

Activity 7: Granting Privileges

GRANT command can be used to grant previliges to the users on tables. In this activity, we are going to demonstrate the use of GRANT command.

The following command grands all type of privileges on *Employees* table to the user *Manager*:

```
GRANT ALL ON Employees TO Manager;
```

Similarly, if we want to give only query access to the *Secretary* then we can write:

```
GRANT SELECT ON Employees TO Secretary;
```

If we want to allow *Manager* to grant privileges to other roles, we use GRANT OPTION as following:

```
GRANT ALL ON Employees TO Manager WITH GRANT OPTION;
```

Activity 8: Revoking Privileges

REVOKE command can be used to revoke previliges granted to the users. In this activity, we are going to demonstrate the use of REVOKE command.

The following command will revoke all the privileges given to *Manager*.

```
REVOKE ALL ON Employees FROM Manager;
```

There are two options available with REVOKE statement: RESTRICT and CASCADE. RESTRICT is the default option. If *Manager* has given privileges to any other user then the previliges can not be revoked through RESTRICT option. In that case CASCADE option needs to be specified and it will cascadedly/transitively revoke privileges from *Manager* as well as from those who were granted previliges by *Manager*.

```
REVOKE ALL ON Employees FROM Manager CASCADE
```

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Lab Task 1

Write a SQL statement that creates an index on *PHONE_NUMBER* attribute of the *Employees* table.

Lab Task 2

Write a SQL statement that creates a view for extracting all the details of employees that belong to *Finance* department.

Lab Task 3

Write a SQL statement that modifies the view created in Lab Task 2 so that it now extracts all the details of employees that belong to *Sales* department.

Lab Task 4

Write a set of SQL statements that creates a user with your own name, assign previliges to select and delete rows in the employee table, select some rows from employees table, revoke all the assigned previliges, and finally try to select rows again.

.

Lab 12

Introduction to MongoDB and JSON; Retreiving Data using MongDB Query API

Objective:

The objective of this lab is to introduce the NoSQL database management system MongoDB. MongoDB uses document-oriented approach to model the data. Specifically, it uses BSON (Binary coded JSON) to model the data. The lab will help students to install the MongoDB server. It will introduce the JSON format and some basic mongo shell commands. It will also familiarize students with the various functions that are part of MongoDB API for retrieving documents based on some predefined criteria.

Activity Outcomes:

After this lab, students should be able to do the following:

- Understand the JSON format
- Use mongo shell to connect to MongoDB server and be able to execute some basic commands
- Search documents based on some criterion
- Apply various comparison operators that can be used in search condition
- Apply various logical operators that can be used in search condition

Instructor Note:

Please get yourself familiarize with the JSON format.

1) Useful Concepts

JSON Format

JSON (JavaScript Object Notation) is an open standard file format and data interchange format that uses human-readable text to store and transmit data objects consisting of attribute–value pairs and arrays. Data is in name / value pairs where a name/value pair consists of a field name followed by a colon, followed by a value:

Example: “name”: “R2-D2”

Data is separated by commas:

Example: “name”: “R2-D2”, race : “Droid”

Curly braces hold objects/document:

Example: {“name”: “R2-D2”, race : “Droid”, affiliation: “rebels”}

An array is stored in brackets []:

Example: [{“name”: “R2-D2”, race : “Droid”, affiliation: “rebels”}, {“name”: “Yoda”, affiliation: “rebels”}]

MongoDB Query API’s *find* Method

To query data from MongoDB collection, you need to use MongoDB's find() method.

Basic syntax of find() method is as follows:

```
db.COLLECTION_NAME.find()
```

In the above syntax, *find()* method without any argument will display all the documents in a non-structured way.

The pretty() Method is another useful method to display the results in a formatted way, you can use pretty() method.

```
db.mycol.find().pretty()
```

2) Solved Lab Tasks

Activity 1: Create a well-formatted JSON document

Suppose there is an employee which has the following field values associated with him:

first_name: John

last_name: Doe

age: 45

hobbies: table tennis, badminton

education: Stanford (Undergrad), MIT (Graduation)

Represent this employee in JSON format.

Solution:

```
{
          “first_name”: “John”,
          “last_name”: “Doe”,
          “age”: 45,
          “hobbies”: [“table tennis”, badminton],
          “education”: {“Undergrad”: “Stanford”, “Graduation”: “MIT”}
}
```

Activity 2: Running some basic commands on mongo shell

Solution:

Basic syntax of use DATABASE statement is as follows:

```
use DATABASE_NAME
```

If you want to create a database with name *mydb*, then *use* DATABASE statement would be as follows:

```
use mydb
```

The following output will be shown:

switched to db mydb

To check your currently selected database, use the command *db*

```
Db
```

The following output will be shown:

mydb

If you want to check your databases list, then use the command *show dbs*.

```
>show dbs

local 0.78125GB

test 0.23012GB
```

Your created database (mydb) is not present in list. To display the database, you need to insert at least one document into it.

```
>db.movie.insert({"name":"Database System-I"})

>show dbs

local 0.78125GB

mydb 0.23012GB

test 0.23012GB
```

In MongoDB default database is test. If you didn't create any database then collections will be stored in test database.

MongoDB db.dropDatabase () command is used to drop an existing database.

```
>show dbs

        local 0.78125GB

        mydb 0.23012GB

        test 0.23012GB
```

If you want to delete the database *mydb*, then dropDatabase() command would be as follows:

```
>use mydb

        switched to db mydb

>db.dropDatabase()

        { "dropped" : "mydb", "ok" : 1 }
```

Now check list of databases

```
>show dbs

        local 0.78125GB

        test 0.23012GB
```

Activity 3: Using find method to display all the documents

```
db.mycol.find().pretty()
```

Output:

```
{
"_id": ObjectId(7df78ad8902c),
"title": "MongoDB Overview",
"description": "MongoDB is no sql database",
"tags": ["mongodb", "database", "NoSQL"],
"likes": "100"
}
```

Activity 4: Using find method to display only one document

```
db.mycol.findOne().pretty()
```

Output:

```
{
"_id": ObjectId(7df78ad8902c),
"title": "MongoDB Overview",
"description": "MongoDB is no sql database",
"tags": ["mongodb", "database", "NoSQL"],
"likes": "100"
}
```

Activity 5: Using find method with a condition

Solution:

To query the document on the basis of some condition, you can use following operations:

```
db.mycol.find({"title":"MongoDB Overview"}).pretty()
```

Activity 6: Using find method with OR condition

Solution:

To query documents based on the OR condition, you need to use $or keyword. Basic syntax of OR is shown below:

```
>db.mycol.find(
{
$or: [
{key1: value1}, {key2:value2}
```

]
}
).pretty()
Example
Below given example will show all the articls written by 'Qasim Malik' or whose title is 'MongoDB Overview'
>db.mycol.find({$or:[{"by":" Qasim Malik "},{"title": "MongoDB Overview"}]}).pretty()
{
"_id": ObjectId(7df78ad8902c),
"title": "MongoDB Overview",
"description": "MongoDB is no sql database",
"by": "Qasim Malik",
"url": "http://www.mywebsite.com",
"tags": ["mongodb", "database", "NoSQL"],
"likes": "100"
}

Activity 7: Using find method with OR and AND together

Solution:

Below given example will show the documents that have likes greater than 100 and whose *title* is either 'MongoDB Overview' or *by* is 'Qasim Malik'. Equivalent sql where clause is 'where likes>10 AND (by = 'Qasim Malik' OR title = 'MongoDB Overview')'

```
db.mycol.find("likes": {$gt:10}, $or: [{"by": " Qasim Malik "}, {"title":
"MongoDB Overview"}] }).pretty()
```

{
"_id": ObjectId(7df78ad8902c),
"title": "MongoDB Overview",
"description": "MongoDB is no sql database",
"by": " Qasim Malik",
"url": "http://mywebsite.com",
"tags": ["mongodb", "database", "NoSQL"],
"likes": "100"
}

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Lab Task 1

Suppose a simple blogging module needs to be incorporated in a website. It will allow bloggers to post their blogs on the website without the need to sign in. Every blog is going to have a unique title, a creation time, a description, a URL, a set of tags assigned by the blogger, a name of the blogger, and the count of total number of likes it receives. Other bloggers can comment on the blog. For each comment, the commentator's name, comment text, data-time and likes it receives, needs to be stored.

a)- Draw an ERD for the above blogging module.

b)- Use document model, and model the same data in a single collection. Provide a sample document of that collection.

Lab Task 2

Write down the API calls along with the number of documents returned for the following needs:

1- *Find out all the listings where property_type is House*
2- *Find out all the listings that have been received more than 100 reviews.*
3- *Find out the number of listings having 8 beds.*
4- *Find the number of AirBnB listings with 3 bedrooms and a review rating greater than 80.*
5- *Find out all the listings that offer Cable TV amenity, accommodate more than 6 individuals, and has moderate cancellation policy*

Lab 13

Manipulating Data using MongoDB API

Objective:

The purpose is this lab is to familiarize students with the various functions part of MongoDB API that allows manipulating the documents. Specifically, it will introduce how to insert, update, save, or delete the documents.

Activity Outcomes:

After this lab, students will be able to:

- Insert new documents
- Import documents in bulk
- Update existing documents
- Save documents
- Remove existing documents

Instructor Note:

1) Useful Concepts

MongoDB CRUD Operations

CRUD operations in MongoDB include *insert, read, update,* and *delete* documents.

Insert Operations

Create or insert operations add new documents to a collection. If the collection does not currently exist, insert operations will create the collection.

MongoDB provides the following methods to insert documents into a collection:

- db.collection.insertOne()
- db.collection.insertMany()

In MongoDB, insert operations target a single collection. All write operations in MongoDB are atomic on the level of a single document.

Read Operations

Read operations retrieve documents from a collection; i.e. query a collection for documents. MongoDB provides the following methods to read documents from a collection:

- db.collection.find()

You can specify query filters or criteria that identify the documents to return.

Update Operations

Update operations modify existing documents in a collection. MongoDB provides the following methods to update documents of a collection:

- db.collection.updateOne()
- db.collection.updateMany()
- db.collection.replaceOne()

In MongoDB, update operations target a single collection. All write operations in MongoDB are atomic on the level of a single document.

You can specify criteria, or filters, that identify the documents to update. These filters use the same syntax as read operations.

Delete Operations

Delete operations remove documents from a collection. MongoDB provides the following methods to delete documents of a collection:

- db.collection.deleteOne()
- db.collection.deleteMany()

In MongoDB, delete operations target a single collection. All write operations in MongoDB are atomic on the level of a single document.

You can specify criteria, or filters, that identify the documents to remove. These filters use the same syntax as read operations.

Bulk Write

MongoDB provides the ability to perform write operations in bulk through *mongoimport* utility.

2) Solved Lab Activities

Activity 1: Using insert method to insert the document

To insert data into MongoDB collection, you need to use MongoDB's insert() or save()method. Basic syntax of insert() command is as follows:

```
db.COLLECTION_NAME.insert(document)
```

Here’s the example inserting a document in the collection:

```
db.mycol.insert({ title: 'MongoDB Overview', description: 'MongoDB is no sql
database', , tags: ['mongodb', 'database', 'NoSQL'], likes: 100 })
```

Activity 2: Using mongoimport tool to bulk insert the documents

MongoDB Provides with a set of tools that can be used to perform various tasks with ease. MongoImport is one such tool. It allows to bulk import the data present in either csv or json file.

Once installed, we can use the following command to insert the documents in bulk.

```
mongoimport --db test --collection restaurants --file d:\file\primer-dataset.json
```

Activity 3: Using update method to update the document

The update() method updates values in the existing document. Basic syntax of update() method is as follows

>db.COLLECTION_NAME.update(SELECTIOIN_CRITERIA, UPDATED_DATA)

Example

Consider the mycol collectioin has following data.

{ "_id" : ObjectId(5983548781331adf45ec5), "title":"MongoDB Overview"}

{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}

Following example will set the new title 'New MongoDB Tutorial' of the documents whose title is 'MongoDB Overview'

```
>db.mycol.update({'title':'MongoDB Overview'},{$set:{'title':'New MongoDB Tutorial'}})
```

```
>db.mycol.find()
```

```
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"New MongoDB Tutorial"}
```

```
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
```

By default mongodb will update only single document, to update multiple you need to set a paramter 'multi' to true.

```
>db.mycol.update({'title':'MongoDB Overview'},{$set:{'title':'New MongoDB
Tutorial'}},{multi:true})
```

Activity 4: Using remove method to remove the document

MongoDB's remove() method is used to remove document from the collection. remove() method accepts two parameters. One is deletion criteria and second is justOne flag

1. deletion criteria : (Optional) deletion criteria according to documents will be removed.

2. justOne : (Optional) if set to true or 1, then remove only one document.

Basic syntax of remove() method is as follows

```
>db.COLLECTION_NAME.remove(DELLETION_CRITTERIA)
```

Example

Consider the mycol collectioin has following data.

```
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"MongoDB Overview"}
```

```
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
```

Following example will remove all the documents whose title is 'MongoDB Overview'

```
>db.mycol.remove({'title':'MongoDB Overview'})
```

```
>db.mycol.find()
```

```
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
```

If there are multiple records and you want to delete only first record, then set justOne parameter in remove() method

```
>db.COLLECTION_NAME.remove(DELETION_CRITERIA,1)
```

If you don't specify deletion criteria, then mongodb will delete whole documents from the collection. This is equivalent

of SQL's truncate command.

>db.mycol.remove()

>db.mycol.find()

Activity 5: Using save method to insert/update the document

The save() method replaces the existing document with the new document passed in save() method. Basic syntax of mongodb save() method is shown below:

```
db.COLLECTION_NAME.save({_id:ObjectId(),NEW_DATA})
```

Following example will replace the document with the _id '5983548781331adf45ec7'

```
db.mycol.save(

{

"_id" : ObjectId(5983548781331adf45ec7), "title":"New Topic"

}

)
```

If you then issue the *find* method call, you will see the following output:

```
db.mycol.find()

{ "_id" : ObjectId(5983548781331adf45ec5), "title":"New Topic"}

{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
```

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Lab Task 1

Please download the AirBnB collection from https://docs.atlas.mongodb.com/sample-data/sample-airbnb/ and import it to Mongo Atlas using mongoimport tool. Paste the screenshot of the terminal showing the command you issued for the import and also the output generated by mongoimport

.

Lab Task 2

Create a collection corresponding to the Employees table of HR schema and insert at least 10 tuples of the table as documents in the collection

Lab Task 3

Update the salary of all the employees by providing a 10% increase

Lab Task 4

Remove all the employees from the Employees collection whose salary is less than 10000.

Lab 14

MongoDB's Aggregation Framework

Objective:

The purpose is this lab is to familiarize students with the aggregate framework provided by MongoDB's API. Grouping and aggregation provides a useful set of operations allowing to extract useful reports from the data.

Activity Outcomes:

After this lab, students should be able to:

- Understand the aggregation pipeline
- Apply various stages of the aggregation pipelines to generate meaningful reports

Instructor Note:

1) Useful Concepts

Introduction

Aggregation operations process data records and return computed results. Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result. In sql count(*) and with group by is an equivalent of mongodb aggregation.

For the aggregation in mongodb you should use aggregate() method. Syntax: Basic syntax of aggregate() method is as follows:

```
db.COLLECTION_NAME.aggregate(AGGREGATE_OPERATIONS_PIPELINE)
```

```
{
_id: ObjectId(7df78ad8902c)
title: 'MongoDB Overview',
description: 'MongoDB is no sql database',
by_user: 'user1',
url: 'http://www.mywebsite.com',
tags: ['mongodb', 'database', 'NoSQL'],
likes: 100
},
{
_id: ObjectId(7df78ad8902d)
title: 'NoSQL Overview',
description: 'No sql database is very fast',
by_user: 'user2',
url: 'http://www.mywebsite.com',
tags: ['mongodb', 'database', 'NoSQL'],
likes: 10
},
```

In UNIX command shell pipeline means the possibility to execute an operation on some input and use the output as the input for the next command and so on. MongoDB also support same concept in aggregation framework. There is a set of possible stages and each of those is taken a set of documents as an input and is producing a resulting set of documents (or the final resulting JSON document at the end of the pipeline). This can then in turn again be used for the next stage and so on.

Possible stages in aggregation framework are following:

- $project: Used to select some specific fields from a collection.
- $match: This is a filtering operation and thus this can reduce the number of documents that are given as input to the next stage.
- $group: This does the actual aggregation as discussed above.
- $sort: Sorts the documents.

- $skip: With this it is possible to skip forward in the list of documents for a given number of documents.
- $limit: This limits the number of documents to look at by the given number starting from the current position.
- $unwind: This is used to unwind document that are using arrays. when using an array, the data is kind of pre-joined and this operation will be undone with this to have individual documents again. Thus, with this stage we will increase the number of documents for the next stage.

2) Solved Lab Activities

Activity 1: Display a list that how many tutorials are written by each user

```
db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$sum : 1}}}])
```

Output:

```
{
      "result" : [
                  {
                        "_id" : 'MongoDB Overview',
                        "num_tutorial" : 2
                  },
                  {
                        "_id" : "NoSQL Overview",
                        "num_tutorial" : 1
                  }
                ],
      "ok" : 1
}
```

Activity 2: Over AirBnB collection, find out the maximum number of beds among all listings.

Solution:

```
db.airbnb.aggregate([{$group:{"_id":null, "MaxBeds":{$max:"$beds"}}}])
```

Activity 3: Over AirBnB collection, find out the minimum price per night grouped by number of beds.

Solution:

```
db.airbnb.aggregate([{$group: {"_id": "$beds", "MinPrice":{$min: "$price"}}}])
```

Activity 4: Over AirBnB collection, find out the number of listings along with average price per night grouped by suburbs and number of bedrooms.

Solution:

```
db.airbnb.aggregate([{$group:  {"_id":  {"Suburb":  "$address.suburb",  "Bedrooms":
"$bedrooms" }, "TotalListings": {$sum:1}, "AvgPrice": {$avg: "$price"}}}])
```

In order to count documents returned as a result of above aggregation, we can add count stage in the pipeline as:

```
db.airbnb.aggregate([{$group:  {"_id":  {"Suburb":  "$address.suburb",  "Bedrooms":
"$bedrooms"  },  "TotalListings":  {$sum:1},  "AvgPrice":  {$avg:  "$price"}}}  ,
{"$count":"TotalDocuments"}])
Total Documents = 971
```

Activity 5: Over AirBnB collection, find out average price per night grouped by room type.

Solution:

```
db.airbnb.aggregate([{$group: {"_id": "$room_type", AvgPrice:{$avg: "$price"}}}])
```

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Lab Task 1

Download the movies collection in json format from https://drive.google.com/file/d/1tAc2PtJzdOWd5Sw9HgHiRPPOpHeaAFfC/view?usp=sharing and import it to MongoDB. How many documents were imported?

Lab Task 2

Using MongoDB aggregation framework over the movies collection, find out the number of movies of each genre. How many movies fall under "Thriller" genre?

Lab Task 3

How many movies in the collection have IMDB rating greater than or equal to 9.5?

Lab Task 4

Which movie has won the most awards?

Lab Task 5

How many movies are there in the collection that belong to Comedy genre, have IMDB rating greater than 8.0, and have won more than 50 awards?